MASTER IN COMPUTER SCIENCE

Tool Support for Design by Contract

Houdart, Adrien

Award date:
2016

Awarding institution:
University of Namur

Université de Namur

Faculté d'informatique

Année académique 2015 - 2016

Tool Support for Design by Contract

Author :
Adrien Houdart

Supervisor :
Pr. Pierre-Yves Schobbens

Mémoire présenté en vue de l'obtention du grade de master en informatique

Abstract

The Skalup development team uses the technique of Design by Contract (DbC) for the realisation of its programs. This methodology aims at introducing the concept of contract in computer applications. These contracts consist of preconditions, postconditions and invariants and are represented in the source code in the form of semi-formal specifications on top of methods and classes.

The technique of Design by Contract was born out of researches conducted by Bertrand Meyer, in connection with its Eiffel programming language. Simultaneously, Liskov has also introduced the concepts of preconditions, postconditions and invariants in her researches on the principle of substitution. Programming with contracts supposes that software developers define precisely what a function / method does and what are the conditions to use it.

Although the DbC technique significantly reduces the number of bugs within an application, Skalup developers face another problem; ensuring that the specifications correspond to the implementation to which they are linked. Currently this check is done manually and therefore takes a considerable time.

The objective of this master thesis is to analyse what exists in terms of code analysis tools for specifications, suggest improvements, contribute to research on this issue and determine the limits of the verifications that can be performed. The second part of the thesis will focus on the realization of a static analysis tool for semi-formal specifications. This tool would aim to inform the developers that an inconsistency between the code and the specification was made in the application.

Keywords: Design by Contract, specifications, precondition, postconditions, invariant, errors detection, static analysis.

Résumé

L'équipe de développement de Skalup utilise la technique de Design by Contract (DbC) pour la réalisation de ses programmes. Cette méthodologie vise à introduire la notion de contrat dans les applications informatiques. Ces contrats comprennent des préconditions, des postconditions ainsi que des invariants et sont représentés dans le code source sous la forme de spécifications semi-formelles en en-tête des méthodes et des classes.

La technique de conception par contrat est née de recherches menées par Bertrand Meyer, dans le cadre de son langage de programmation Eiffel. En même temps, Liskov a également introduit les concepts de préconditions, postconditions et invariants dans ses recherches sur le principe de substitution. La méthode de Design by Contract suppose que les développeurs de logiciels définissent précisément ce qu'une fonction ou méthode fait et quelles sont les conditions à respecter pour pouvoir l'utiliser.

Bien que la technique DbC réduit de manière significative le nombre de bugs dans une application, les développeurs de Skalup font face à une autre problèmatique ; faire en sorte que les spécifications correspondent à l'implémentation à laquelle elles sont liées. Actuellement, cette vérification est effectuée manuellement et prend donc un temps considérable.

L'objectif de ce mémoire est d'analyser ce qui existe en termes d'outils d'analyse de code pour les spécifications, suggérer des améliorations, contribuer à la recherche sur ce problème et de déterminer les limites des vérifications qui peuvent être effectuées. La deuxième partie de ce mémoire se concentrera sur la réalisation d'un outil d'analyse statique pour les spécifications semi-formelles. Cet outil a pour but d'informer les développeurs qu'une incohérence entre le code et les spécification a été introduite dans le programme.

Mots clés : Design by Contract, spécifications, précondition, postconditions, invariant, détection d'erreurs, analyse statique.

I would first like to thank my thesis advisor Pierre-Yves Schobbens, who read my numerous revisions and steered me in the right direction when I needed it, but also for its availability, its advices and its support for the realization of this thesis.

I would also like to acknowledge my colleagues, Marco Willemart and Jean-Marc Davril who gave me very valuable advices for this master thesis.

Finally I thank my family and friends who supported me during the realization of this master thesis.

CONTENTS

1 Introduction **1**

2 Background and Related Work **3**

2.1 Design by Contract 3

2.1.1 Eiffel 4

2.1.2 The Daikon invariant detector 8

2.1.3 Java Modeling Language (JML) 8

2.1.4 Modern Jass 10

2.1.5 Spring Contracts 11

2.1.6 Contract for Java (C4J) 11

2.1.7 Code Contracts 12

2.1.8 Cofoja 13

2.1.9 Bean Validation 14

2.1.10 Valid4j 15

2.1.11 Contract4j 15

2.1.12 Comparison 17

2.1.13 Conclusion 20

2.2 Static code analysis 21

2.2.1 Checkstyle 21

2.2.2 FindBugs 22

2.2.3 PMD 22

2.2.4 Jtest 23

2.2.5 SonarQube 23

2.2.6 Coverity 24

2.2.7 GrammaTech CodeSonar 25

2.2.8 Comparison 26
2.2.9 Conclusion 27

3 Specification by Skalup **28**

3.1 Class and method specifications 28
3.1.1 Abstract Values and Abstract State 30
3.1.2 Method Specifications 34
3.2 Abstraction Functions and Representation Invariants 39
3.2.1 Abstraction and concrete representation 41
3.2.2 Abstraction function 42
3.2.3 Representation Invariants 44
3.3 Semi-formal specifications 46

4 Research and contribution **47**

4.1 Research on potential static verifications 47
4.1.1 Class and method specifications 47
4.1.2 Abstraction Functions and Representation Invariants . . 54
4.1.3 Summary 58
4.2 Prototype of the specifications analysis tool 60
4.2.1 The technology 60
4.2.2 Choice of the analyze engine 61
4.2.3 The Application Programming Interface 63
4.2.4 The implemented tests 66
4.2.5 Summary 82

5 Conclusion **84**

5.1 General conclusion 84
5.2 Future Work 86

Bibliography **90**

A Appendices **91**

A.1 DataType.java 92

A.2 InnerDataType.java 96
A.3 Modifier.java 97
A.4 ModifierType.java 100
A.5 Specfield.java 103
A.6 SpecfieldRef.java 106
A.7 SpecfieldCheck.java 107
A.8 ModifiesCheck.java 115

LIST OF FIGURES

2.1 Modern Jass illustration[1] 10

3.1 The *Segment* class, example from [2] 29
3.2 The specification of the *Circle* ADT 31
3.3 The specification of the *Circle* with derived fields 32
3.4 The *Circle* class 34
3.5 Example of usage of assertions 36
3.6 Example of usage of spec fields for specifications 37
3.7 Example of usage of derived fields for specifications 38
3.8 *Segment* class with its implementation 40
3.9 Abstract function for the *Segment* class 40
3.10 *Complex* ADT 41
3.11 R and A 42
3.12 Abstraction Function 42
3.13 Abstraction Function bis 43
3.14 AF one-to-one correspondence 43
3.15 RI example 1 44
3.16 RI example 2 44
3.17 RI example 3 44
3.18 RI example 4 44
3.19 Complex example of a Abstraction Function 45

4.1 Exposition of the representation 50
4.2 Correction of the exposition of the representation 51
4.3 Example of representation invariant 56
4.4 Example of more complex representation invariant 57
4.5 *Specfield* class specification 67
4.6 *SpecfieldRef* class specification 68
4.7 *DataType* class specification 70
4.8 *SpecificationDataType* class specification 71
4.9 *InnerDataType* class specification 72
4.10 Duplicate specification fields 73
4.11 Unknown specification fields 74
4.12 *Method* class specification 76
4.13 *ModifierType* class specification 78
4.14 Missing @*effects* clause 78
4.15 Mutator method on immutable class 79
4.16 Mutability changed through inheritance 79
4.17 Missing @*modifies* clause 80
4.18 Instance variable modified in immutable class 80
4.19 Method defined on the super class 80
4.20 Method implemented in the sub class 80

A.1 *DataType* class 95
A.2 *InnerDataType* class 96
A.3 *Modifier* class 99
A.4 *ModifierType* class 102
A.5 *Specfield* class 105
A.6 *SpecfieldRef* class 106
A.7 *SpecfieldCheck* class 114
A.8 *ModifiesCheck* class 122

CHAPTER 1

INTRODUCTION

This master thesis concentrates on the study of how developers use specifications to detail their source code within the development team of Skalup. Based on this, different verifications will be detailed to prevent the introduction of unconformities between the specifications and the implementations in the codebase of the development team.

Design by Contract (DbC) consist of defining contracts that must be respected by programmers that use methods and classes developed by another programmer. In turn, the programmer who defined the specification must guarantee that its program does what it is supposed to do. The conditions that have to be respected by the caller are the *preconditions* and the ones that have to be hold true by the developer of the methods are called *postconditions*.

The Design by Contract approach is used by the developers of Skalup. DbC separates the implementation from the specifications. The goal of the specifications is to define what the program does and the goal of the implementation is to described how it is done.

Although Design by Contract significantly reduces the number of bugs within an application, it may introduce new kind of inaccuracies; specification errors. Currently, reviews of the code are made manually to eliminate these errors. However, developers are not infallible and may miss errors. Moreover, this approach takes considerable time and requires developers to spend time on error detection at the expense of developing new functionalities.

The master thesis aims to:

- present the state of the art in order to understand the existing tools in the area of Design by Contract and static code analysis;
- analyse the usage of the specifications in the development team of Skalup;
- detail verifications to avoid introducing errors in specification or implementation of programs;
- make a prototype to implement the previously detailed checks.

The contribution of this thesis is to provide a mechanism to easily implement checks that will verify the implementations and the specifications provided by programmers to ensure the accuracy of the source code of a program.

Finally, we conclude this thesis by analysing the different tests found and the checks that have been integrated into the prototype. We also suggest depending of the realized tests, ideas to continue the study of the detection of errors in specifications.

Finally, here is the structure of this document:

— First, different technologies, addressing the specifications and the static analysis of source code will be analysed. The researches conducted about this topic will be analysed to enforce the cases covered by this thesis.

— Secondly, the usage of specifications among the development team of Skalup will be detailed to help understanding how they work, and how developers could introduce errors in the codebase.

— Then, the fourth chapter will detail the different tests that could be perform to detect erroneous specifications or discordance between the specifications and the implementation they represent. This chapter will also detail a prototype realized as part of this thesis, which implements different tests to detect errors in specifications.

— Finally, the last chapter will conclude this thesis by identifying strengths and weaknesses for the verifications. It will also suggest improvements that could be done in future work.

— The annexes

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter will explains the general concepts used during this thesis. In particular Design by Contract (DbC) will be described as well as the static analysis of source code. Tools using these concepts, or similar concepts will also be detailed.

2.1 Design by Contract

Design by Contract (DbC), also known as programming by contract, contract programming and design-by-contract programming, is a mechanism that prevents 90% of the bugs from occurring [3]. This approach, used for designing programs, forces developers to define specifications for the interfaces they create in their programs. These specifications are made of preconditions, postconditions and invariants. Their goal is to describe the abstract data type of the interface they are designing. These specifications constitute contracts. The DbC approach assumes that the client and the supplier of an interface are aware of the contract and will respect them (all preconditions must be respected before using the interface) [4]. The programmer considers preconditions are valid at all time and does not need to verify them. If a precondition is violated, the program should fail. Nevertheless, it is a good practice to defensively assert that all preconditions are hold true. This approach is called defensive programming. In this case, if a precondition is not respected, the program should throw an exception to inform the client that the precondition was not respected. Both approaches help the client to figure out where the precondition was broken and help him to fix its code [4].

If the client respects its part of the contract (preconditions), the supplier ensures that the postconditions will be satisfied as well.

Design by Contract comes from researches conducted by Bertrand Meyer starting in 1986 [5]. It is deeply integrated in its language, Eiffel. During the same time, Barbara Liskov, known for the Liskov Substitution Principle (LSP) made some researches that led to the same results as Design by Contract [6]. Indeed, the Liskov Substitution Principle is strongly related to pre and postconditions.

The concept of contracts is applicable to methods and procedures and will

contain the following elements.

— The valid and invalid inputs, their types and their meaning
— The return value of the method (its type and its meaning)
— The Exception(s) (errors) that may occur and the reason
— The side effects
— The Preconditions
— The Postconditions
— The Invariant

In object oriented programs, classes can be subclasses and thus, specifications are inherited. However, these specifications can be made more precise. For a subclass, the preconditions can be weakened but never strengthened. Postconditions can be strengthened but never be weakened [6].

"Design by Contract revolutionizes software construction by weeding out bugs before they get the opportunity to harm the software", eiffel.com.

2.1.1 Eiffel

Eiffel is an oriented object programming language (OOP) designed by Bertrand Meyer [7]. This is a language made for the concept of programming by contract.

Several concepts introduced by Meyer in its language, Eiffel, have been integrated in C# and Java. Eiffel keeps-on evolving and integrates new innovative concepts [3].

An IDE (Integrated Development Environment) for the Eiffel language exists. It is called EiffelStudio. It permits, to edit, compile and debug programs.

The name Eiffel comes from the French engineer Gustave Eiffel, who made the Eiffel tower. Gustave Eiffel manage to build the tower within the budget he received and in the allotted time which is the goal of the Eiffel language. Most of the time, projects don't respect the budget and the due dates, so the name of the language is a nod to the objective of respecting delay and budget in software development.

The main goal of the Eiffel programming language is to let programmers create reliable and reusable softwares. The language supports multiple aspects of the OOP such as multiple inheritance, polymorphism, encapsulation, genericity, type-safe conversion, etc.

Eiffel contributes a lot to software engineering, but its most important contribution is Design by Contract, which included the concepts of assertions,

preconditions, postconditions and class invariants in a programming language. All these concepts are used to ensure the correctness of the program without affecting efficiency.

The Eiffel language is based on DbC, contracts that are defined with formal specifications. Moreover, it provides automatic and high quality documentation. The specifications provide the ability to communicate the design of the application being made without having to describe the implementation details.

The following source code represents a mathematical circle. It is described using the Eiffel language and its specifications.

```
class
    CIRCLE

inherit
    POINT
        rename
            make as point_make
        redefine
            make_origin,
            out
        end
create
    make, make_origin, make_from_point

feature -- Initialization

    make (a_x, a_y, a_r: INTEGER)
            -- Create with values ‘a_x’ and ‘a_y’ and ‘a_r’
        require
            non_negative_radius_argument: a_r >= 0
        do
            point_make (a_x, a_y)
            set_r (a_r)
        ensure
            x_set: x = a_x
            y_set: y = a_y
            r_set: r = a_r
        end

    make_origin
            -- Create at origin with zero radius
        do
            Precursor
        ensure then
            r_set: r = 0
        end
```

```
    make_from_point (a_p: POINT; a_r: INTEGER)
            -- Initialize from ‘a_r’ with radius ‘a_r’.
        require
            non_negative_radius_argument: a_r >= 0
        do
            set_x (a_p.x)
            set_y (a_p.y)
            set_r (a_r)
        ensure
            x_set: x = a_p.x
            y_set: y = a_p.y
            r_set: r = a_r
        end

feature -- Access

    r: INTEGER assign set_r
            -- Radius

feature -- Element change

    set_r (a_r: INTEGER)
            -- Set radius (‘r’) to ‘a_r’
        require
            non_negative_radius_argument: a_r >= 0
        do
            r := a_r
        ensure
            r_set: r = a_r
        end

feature -- Output

    out: STRING
            -- Display as string
        do
            Result := "Circle: x = " + x.out + " y = " + y.out + " r =
                " + r.out
        end

invariant

    non_negative_radius: r >= 0

end
```

The source code shown above describes different methods for the Circle class such as:

— make(...)
— make_origin
— make_from_point(...)
— set_r(...)

which are mutator methods. Each method has its own pre and postconditions.

There is also an invariant, represented at the end of the source code by **non_negative_radius: r >= 0**, forcing the radius **r** to be greather than or equals to 0.

2.1.2 The Daikon invariant detector

An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications, Michael D. Ernst, Jeff H. Perkins, The Daikon system for dynamic detection of likely invariants [8].

Daikon is an invariant detector that runs on a given source of data, typically a run-time log of a program. Daikon analyses data computed by a program, summarizes the observed valued and extracts invariants from these values. Daikon can run on various languages such as C, C++, C#, Eiffel, F#, Java, Perl and Visual Basic. It also works with spreadsheets and other data sources. Specifications and invariants are helpful to understand, construct and maintain programs [9].

Invariant can be either used by a human or by a tool. Humans will use it for documentation purpose in order to prevent the introduction of bugs (a well documented invariant will prevent the violation of this invariant, on the contrary, poor/non documented invariant may lead to introduction of bugs). Invariant are also used by humans to highlight unusual conditions that may require more attention **by** the programmer. Tools use invariant to check hypothesis by converting the invariants into assertions.

Daikon examines computed values at runtime, and looks for relationship among the values. The examinations of those values may lead to false positives (illegitimate properties) or to false negatives (missing properties).

This tool is more accurate when running on a particular part of a program and not on the whole program. Running Daikon on a big set of data will result in an overwhelming of data for the tool itself or for the person who will analyze the outputs [9].

There is no information about the community and how many projects are using the Daikon invariant detector. However, new versions of the program are frequently released, around one each month.

2.1.3 Java Modeling Language (JML)

JML is a formal specification language for Java application. It provides a semantics for the purpose of formally describing how a Java application behaves. Its goal is to remove all ambiguities that may have been introduced by the developers. JML can be compared with the Eiffel language. Its purpose is to provide a rigorous formal semantic verification, remaining accessible to any Java developer[10]. The specifications can either be written as annotations

in the comments or be written in a separated file. These specifications can be used by tools for analysis purpose. In addition, a JML program can be compiled by any Java compiler because the specifications are written in the comments of the program and thus have no impact on the compilation [11].

JML supports many specifications such as

- Requires: precondition
- Ensures: postcondition
- Signals: define a condition when a particular exception may be thrown
- Assignable: defines attributes that may be assignable in the method
- Pure: declares a method that does not modify any attributes, with no side effect (syntactic sugar for assignable \nothing)
- Invariant: defines an invariant for the class
- Also: adds additional specifications to inherited specifications ()
- Assert: defines JML assertions

JML provides also some basic expressions:

Annotation	**Meaning**
\result	identifier for the returned value
\old(<name>)	allow to refer the value of the variable <name> when calling the method
\forall	universal quantifier
\exists	existential quantifier
a => b	the logic relation a implies b
a <=> b	the logic relation a equivales b
&&	logic AND
||	logic OR
!=	logic NOT

JML annotations can also access Java objects, their methods, etc.

There are plenty of tools that use these specifications. OpenJML is a program that integrates static verifications for JML into Java programs. It supports Java 1.5+ and it is the successor of ESC/Java2 which only support Java 1.4 [12]. Iowa State JML is another tool that can convert the JML annotations into assertions. It can also generate Javadoc for the applications. This Javadoc contains all the specification defined by JML and make a 'improved' Javadoc for the program. This means that the Javadoc is richer with JML than the traditional Javadoc. This tool can also generate unit tests from the specifications.

2.1.4 Modern Jass

Modern Jass also integrates the concept of Design by Contract in Java applications. It uses the annotations introduced in Java 1.5 to specify the contracts.

Johannes Rieken presents in his thesis the design and implementation of a DBC-tool for Java. He proposes a revised version of what JML does. Its approach uses Java Annotations and the Instrumentation API to use Design by Contract with Java. In contrast to existent Design by Contract tools, his implementation integrates seamlessly with Java is easy to maintain [13].

Several annotations are defined by Modern Jass to allow the programmer to define the behavior of its program such as:

— @Pre: precondition
— @Post: postcondition
— @Also: container for multiple specifications
— @NonNull: state that a reference is not null
— @Min: define the lower bound of a numerical value
— etc.

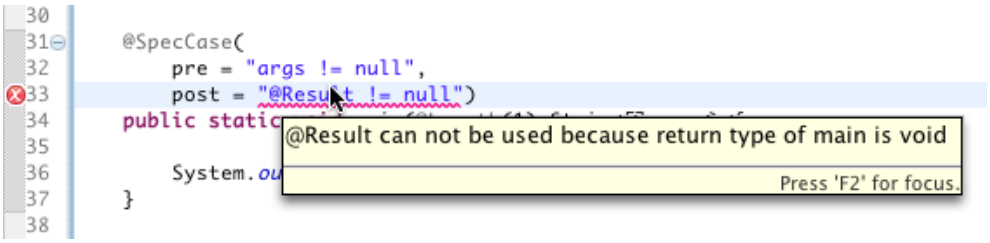

Figure 2.1 – Modern Jass illustration[1]

These annotations are analyzed and validated by Modern Jass. Indeed, the Java annotation are accessible by the Java compiler API (Java 1.6+) and can then be used by Modern Jass to perform static analysis. The use of annotations makes the specifications available in the AST (Abstract Syntax Tree). In Contrast, using comments/Javadoc for the specifications makes it more difficult to analyze the specifications because the comments have to be parsed to fetch the contracts. Modern Jass integrates with the main Integrated Developer Environments (IDEs) and can then show the results of static analysis to the programmer [1].

The specifications are also enforced when a program is executed. To do so, Modern Jass adds assertions at runtime and stops the execution if an assertion is violated. In order to add these assertions, Modern Jass uses the Bytecode Instrumentation API and so, modifies the bytecode of the application.

The project does no longer evolve. The last update was released on the 08th

of April 2013.

2.1.5 Spring Contracts

Spring Contracts is an add-on to the Spring framework. It provides a solution to integrate the concept of Design by Contract into a Spring application. As many specification frameworks, it uses the Java Annotation to describes the preconditions, postconditions and invariants.

It supports a few features such as [14]:
- Annotations for describing the specifications (on classes, constructors and methods
- Inheritance of contracts (deep inheritance)
- Recognition of contracts on Interface (multi-inheritance)
- Custom messages for contracts violation
- Define behavior on how to react when a contract is violated
- etc.

The project seems to be abandoned. The last release was in January 2007 in an early version (0.3).

2.1.6 Contract for Java (C4J)

Contract for Java or C4J is also a framework that provides facilities to integrate contracts in a Java application.

It was developed by Jonas Bergstrom and later used by Hagen Buchwald at the Karlsruhe Institute of Technology (KIT) for eduction purpose.

Unlike other frameworks that mostly use Java Annotations, C4J uses a separate class which gathers the specifications of a class. A link between the class and its contract class is made by using the annotation @Target.

They point to the fact that classes will stay nice and clean to justify the use of separate files for contracts.

Developers can use the power of Java to describe specifications with C4J, because the contracts are plain Java classes. Thus all the Java features can be used to describe specifications, however complex there are.

Like the other frameworks, C4J supports
- Preconditions
- Postconditions
- Class invariants

— Access to the old value in postconditions
— Contracts on interfaces and classes
— Inheritance of contracts
— Extension of contracts (for inherited contracts)
— etc.

Unfortunately, this project no longer evolves. The last version was release in november 2012.

2.1.7 Code Contracts

Code Contracts provide a way to apply DbC in .Net programs. Code Contracts is a tool developed by Microsoft Research. The contracts are made of preconditions, postconditions and object invariants as in other tools. The contracts are used to perform static verifications, documentation generation and runtime checking. Microsoft Research has published Code Contracts as an open source tool and it takes part in the Open Source for Academics program [15].

Code Contracts works out of the box with all .NET framework and integrates in Visual Studio. The contracts take the form of assertions directly written in the body of the methods.

Preconditions are expressed with the following statement, for instance:

```
Contract.Requires(x != null);
```

If the code is supposed to throw a particular exception on precondition violation, the following statement is used:

```
Contract.Requires<ArgumentNullException>(x != null, "x");
```

where *ArgumentNullException* is the type of exception being thrown in case of precondition failure, and the string "x" is the name of the variable being null in this case.

It is also possible to define blocks of contracts. It allows to extend the existing checks with custom checks defined in theses blocks.

```
if (x == null) throw new ...
Contract.EndContractBlock(); // All previous "if" checks are
    preconditions
```

Postconditions are defined in the same way as preconditions except that the Ensures method is used:

```
Contract.Ensures(this.F > 0);
```

It is also possible to refer to the result value of the method in the postcondition by using the Result method.

```
Contract.Ensures(0 < Contract.Result<int>())
```

Code Contracts defines other methods programmers can use when defining the contracts such as

- Contract.OldValue
- Contract.ForAll
- Contract.Exist
- Contract.Invariant

2.1.8 Cofoja

Cofoja stands for Contract for Java and is a contract programming framework. It uses Java annotation and the instrumentation API of Java to perform runtime verifications to ensure that contracts are respected. This framework was created by Nhat Minh Lê when he worked at Google Switzerland as an intern back in 2010 [16]. Its work is based on Modern Jass. It is currently maintained by Nhat Mink Lê and a small community on Github. The contracts are written as quoted strings in annotations. E.g. *@requires("x <= 10")* represents a precondition that requires that x is lower or equals to 10. Any Java expression can be used in the annotation provided by the framework, exept anonymous classes.

There are three main annotation provided by Cofoja:
- @requires : used to define preconditions
- @Ensures : used to define postconditions
- @Invariant : used to define the class invariants

These annotation are written before the methods and they can be combined as shown below.

```
@Requires("x >= 0")
@Ensures("result >= 0")
static double sqrt(double x);
```

As shown in the previous example, the precondition can refer to parameters of the method it is attached to. Pre and post conditions are evaluated as if they was in the scope of the function and so have access of the same parameters/-

variables as if they were written in the method. In addition, the postconditions can use special keywords to refer to specific values such as: *result* to refer to the returned value, *old* to refer to the value of a variable when the method was entered (e.g. *old (x)*).

It is also possible to define invariants with the @*Invariant* annotation. The invariants are checked on entry and on exit of method calls and ensure that the methods keep the invariants valid through the lifecycle of an object.

Cofoja also supports inheritance of specifications. A subclass can augment the specifications defined on its superclass but has to respect them (preconditions can be weakened and postconditions can be strengthened). Cofoja checks either that the preconditions of the superclass are respected or the preconditions of the subclass, but not both. For postconditions, Cofoja checks both, the specifications from the super class and from the subclass.

There exists another annotation, @*ThrowEnsures* that is used to guarantee that the state of an object stay valid on abnormal method exits (when exceptions are thrown by the method). For normal method exit, the @*Ensure* annotation guarantee the validity of the state of the object [17].

2.1.9 Bean Validation

Bean Validation is not a Design by Contract oriented framework but rather a validation framework that provides facilities to validate constraints on Java Beans. This framework make use of the Java annotations to let programmers express their constraints on their Beans. There are preexisting annotations such as; @*NotNull*, @*Email*, @*Min*(...), *Max*(...), *Size*($min = ..., max = ...$), etc. that are used to describe the range of valid input for a given Java Bean [18].

It is possible to extends the existing constraints with custom ones.

The framework provide an API to validate the object against theses constraints. If some constraints are violated, the set of violations can be retrieved and error messages (localized) can be displayed. This API is integrated in Java EE since the version 6.

This framework works differently than other frameworks/tools described in this chapter, because it is a validation framework, its goal is to validate fields on objects and retrieve error messages and not the ensure that pre and postconditions are valid on these objects [19].

2.1.10 Valid4j

Valid4j is a assertion and validation library that provides useful methods to express pre- and postconditions when programming with Design by Contract in Java [20]. The preconditions are expressed with the Java method $require(..., ...)$ where the first parameter is the variable that will be 'checked', and the second is the expression used to 'check' the variable. Example:

```
require(list, everyItem(greaterThanOrEqualTo(3)));
```

or for more simple preconditions, the *ensure* method takes also only one parameter;

```
ensure(result != null);
```

The *require* method returns the given input, and ensures that it is a valid input, so the following code can be used to ensure that the given *message* variable contains a predefined string, and at the same time, initialize the message member;

```
this.message = require(message, containsString("Greetings"));
```

Valid4j uses the same principle to express postconditions, using a method called $ensure(...)$.

```
return ensure(list, hasSize(greaterThan(1)));
```

The previous code returns the variable *list*, and the *ensure* methods ensures that the post conditions is respected by the method.

There are also some other useful predefined assertions such as $neverGetHere(...)$ that asserts that this part of the source code is never reached.

If an assertion is violated, an exception is thrown, preventing the code to be executed when being in an incorrect state [20].

2.1.11 Contract4j

Contract4j is a Design by Contract tool for Java applications. The contracts are expressed using Java annotations. The contracts are then evaluated at runtime by the tool, using Aspect-Oriented Programming.

Specification are written on classes or interfaces so that clients can see these

constraints. AOP is then used to test the constraints at runtime without cluttering the source code with logic to verify that pre- and posts-conditions are respected. [21]

Different annotations are predefined to express contracts;

— @Pre("...") annotation used to define the preconditions
— @Post("...") annotation used to define the postconditions
— @Invar("...") annotation used to define the class invariants

In case of contract violation, Contract4j will throw an exception that inherits from the *ContractError* class. This will inform the developer that an error occurred.

Contract4j provides a minimal support for inheritance of specifications because Java annotations are not inherited from a superclass. So developers have to rewrite the annotation of the parent on the subclass.

2.1.12 Comparison

The following table summarizes and compares the different technologies working with specifications described earlier.

	Eiffel	Daikon	OpenJML	Modern Jass	Spring Contracts	C4J
Languages	Eiffel	Java, C, C++, Eiffel, Perl, C#, F#	Java	Java	Java	Java
Specification form	Part of the language	n/a	Javadoc Annotation or other file	Java annotations	Java annotations	Separated file
License		MIT	Opensource	LGPL	Opensource	EPL
Current Version	4.2	5.2.24	0.6.3		0.3	6.0.0
Latest release		March 2016	April 2015	April 2013	Jannuary 2007	Nov 2012
Precondition	require	n/a	Requires	@Pre	yes	assert
Postcondition	ensure	n/a	Ensures	@Post	yes	assert
Invariant	invariant		Invariant		yes	@ClassInvariant
Inheritance	yes	n/a	Also keyword	@Also annotation	Deep inheritance	yes
Result	Result	n/a	\result			
forall		n/a	\forall			
exists		n/a	\exists			
old	old	n/a	\old(...)			old(...)
AF		n/a				
RI	*invariant*					
Type of analysis	At run-time		At run-time	At run-time	At run-time	At run-time

	Code Contracts	Cofoja	Bean Validation	Valid4J	Contract4J
Languages	.NET (C# & VB)	Java	Java	Java	Java
Specification form	Assertions in .NET	Java annotations	Java annotations	Assertions	Java annotations
License	MIT	LGPL	Apache 2.0	Apache 2.0	Apache 2.0
Current Version	1.9.1	1.3	1.1	0.5.0	0.9
Latest release	July 2016	February 2016	April 2013		April 2013
Precondition	Contract.Requires	Requires("")	n/a	require method	Pre
Postcondition	Contract.Ensures	Ensure(""), ThrowEnsures("")	n/a	ensure method	Post
Invariant		Invariant("")	n/a	n/a	Invar
Inheritance		yes	n/a	n/a	yes but limited
Result		Supported through Java expression	n/a	n/a	n/a
forall		Supported through Java expression	n/a	n/a	n/a
exists		Supported through Java expression	n/a	n/a	n/a
old		Supported for postconditions	n/a	n/a	n/a
AF		No	n/a	n/a	n/a
RI		Yes through the Invariant("")	n/a	n/a	Yes through the Invar("")
Type of analysis	At run-time	At run-time	At run-time	At run-time	At run-time

2.1.13 Conclusion

All the tools described above are interesting and permit to combine Design by Contract with programming. However, the specifications used at Skalup are very specific. None of the previous tool support the same specifications as the ones used by Skalup. This makes impossible to use out of the box one of the tools detailed earlier. Nevertheless, these tools are great sources of inspiration and constitute a starting point for a tailor-made tool.

2.2 Static code analysis

The goal of static analysis tools is to analyze the source code of a program without executing it (as opposed to, dynamic analysis which is the analysis of a program while executing it). It is performed by an automated tool.

The analysis of the source code of a program performed by a human is called a code review, a program understanding or a program comprehension.

One possible application of static analysis is to use it for automated debugging purpose, especially for runtime errors. The most common mistakes a static analyse can detect includes uninitialized or undeclared variables, circular references, and typos.

The static analysis of programs is undecidable. There is no method that can always tell truthfully if a program will produce runtime errors or not. This is a mathematical result based on the research conducted by Alan Turing and described in the "Halting Problem"[22] and by Henry Gordon Rice in the "Rice's theorem"[23]. As a result, a static code analysis may result in false positives or false negatives.

2.2.1 Checkstyle

Checkstyle is a syntactic analysis tool used to check that the source code of a Java Application follows some coding rules. The tests are limited, they cannot analyse the contents of variables or assume that the program is correct or complete. It cannot find bugs either.

Each checking rule can raise messages (notifications, warnings or errors) when they are violated.

Checkstyle can analyze multiple features of an application such as:
- Javadoc comments
- Naming conventions
- Limit the number of parameters in functions or methods
- Limit the length of a line
- Check for mandatory headers
- Check the spaces between characters and check formatting
- Detect duplication of code
- Calculate the complexity of a methods or a class
- etc.

The checks performed by Checkstyle are oriented towards respect for conventions. These tests can be extended with custom checks. Checkstyle provide a

rich API for easily create new checks [24]. New releases are published every month and there are 80 contributors on their Github which demonstrates that the project is continuously evolving [25].

2.2.2 FindBugs

FindBugs is also a static code analyzer that is oriented, as its name suggests, towards the detection of potential bugs in Java code. FindBugs operates on Java bytecode rather than on the source code. It can be integrated in other tools such as Eclipse, NetBeans, IntelliJ, Jenkins, etc. It is developed by three main developers and has 30 others contributors on Github. The project is supported by important IT companies such as Google and Sun Microsystem. New releases are publish every three to six months.

It is possible to add custom rules to increase the amount of tests performed on the application.

The strengths of FindBugs are that it often finds real problems and has a low rate of false positive detections. Findbugs runs on the bytecode of a program, whicht makes the executions of the verifications being fast. On the negative side, FindBugs needs compiled code and is not aware of the source code.

2.2.3 PMD

PDM is another static analysis tool for Java source code (it supports many other languages such as JavaScript, PLSQL, XML, C, C++, C#, PHP, Fortran, Scala, etc.). It has predefined rules that check the quality of a program such as

- Useless code
- Imbrication complexity
- Detection of duplicate code
- Detection of dead code (code that will never be executed)
- Detection of unused variables
- Detection of empty catch blocks
- etc.

This tool can also be integrated with other tools. There are official plugins for Maven, Eclipse, NetBeans, JBuilder, JDeveloper and IntelliJ IDEA.

The checks performed by PMD are oriented towards the detection of bad practices.

New versions of PMD are released frequently. The average is one every two months.

2.2.4 Jtest

Jtest, as its name suggests, is a testing software for Java applications. But it is also a static analysis tool. This program is made by Parasoft. It includes features for code review, unit test generation and execution, regression testing, static analysis, runtime error detection and Design by Contract.

It also supports a broad range of languages such as C, C++, Java, C#, etc.

Its static analysis is capable of:
- Measuring multiple metrics on the source code
- Optimizing unit tests in order to increase their coverage
- Integrating with other static analysis tools

JTest received multiple award for "Best Software Testing Solution" from the Software and Information Industry Association (SIIA) [26]. It also received an award for the "Technology of the Year" from InfoWord [27].

JTest is capable of detecting some defects in a program such as:
- API usage errors
- Best practice coding violation
- Hierarchy inconsistencies
- Deadlocks
- Memory corruption and illegal accesses
- Null pointer references
- Rule violations
- Best practice for security violation
- SQL injection
- etc.

JTest does not require to use DbC to work, however, using Design by Contract increases the potential of JTest. Indeed, JTest can automatically create black-box tests to verify that contracts are respected but it will not generate tests that violate the preconditions defined for the methods. There exists another tool called JContract that runs in parallel with JTest. It checks that the contracts are not violated at runtime.

2.2.5 SonarQube

SonarQube (more formerly Sonar), is an open source platform used for continuous inspection of the quality of the source code. It integrates three popular static analysis tools to analyse source code: Checkstyle, PMD and FindBugs. It comes with many rules that can prevent potential runtime errors, improve code quality, increase unit test coverage, enhance the design and architecture etc. SonarQube has also a GUI (Graphical User Interface) that summarizes

metrics collected during analyses. It also shows other measurements such as the number of lines of code, the test coverage or the complexity of each class. These measurements are stored in a way to see the evolution of the quality of the program over time.

SonarQube is a tool that can be used out of the box with a rich variety of predefined rules, but these rules can also be extended with custom rules. SonarQube will execute multiple plugins such as CheckStyle, FindBugs and PMD. There are also many other plugins that can be run on the source code for static analysis purpose. As a result, SonarQube takes advantage of the strength of each plugin to perform the analysis in view of getting feedbacks on the quality of the code. SonarQube has also developed its own analysis library, called SSLR, standing for SonarSource Language Recognizer. SSLR provides tools to easily create quality rules or compute measures by creating a lexer, a parser and an AST visitor.

Moreover, the main added value is the fact that SonarQube records and stores metrics of the analysis in a database, so the quality trend over time can be observed by the development team. The metrics can indicate whether the quality of the application is getting better or worse. SonarQube also works with continuous integration tools such as Atlassian Bamboo, Jenkins, Hudson, etc. and supports more than 25 languages among which Java, C, C++, C#, PHP, Python, etc.

2.2.6 Coverity

Coverity is a static code analysis tool that works with C, C++, Java, C# and JavaScript sources. That tool was originally founded by the Computer Systems Laboratory at Stanford University in Palo Alto. In 2008, Synopsys (electronic design automation company) acquired it. This tool helps developers to troubleshoot and fix critical bugs in their programs. Coverity was awarded some prices; its CEO was named in 2008 by MIT Technology Review Magazine as *One of World's Top Young Innovators Under the Age of 35* [28], in 2009 and 2011, Coverity was named *One of the Fastest Growing Companies in North America* on Deloitte's Technology Fast 500^{TM} [29] [30], and Covery won the CODiE Award for *Best Software Development Solution* in 2012 [31]. This tool is widely used in the open source community [32] and is currently used by Linux, PostgreSQL, OpenSSL, Firefox and many more.

2.2.7 GrammaTech CodeSonar

Grammatech is an American company specializing in tools for software development and was founded in 1988 within Cornell University. CodeSonar is a static analysis tool for C and C ++ source code, that detects programming errors at runtime. Its goal is to detect security vulnerabilities in order to drastically reduce the number of latent errors in software and increased robustness, such as Null pointer dereferences, divides by zero, uses after free, initialized variables, unreachable code, misused of memory allocation and many more. CodeSonar works concurrently with a C/C++ compiler (such as Microsoft Visual Studio) and analyses the program when it is running. When the compilation is completed, the generated files are used by CodeSonar to synthesize a model that will be used to execute the analysis [33].

2.2.8 Comparison

The following table summarizes and compares the different technologies of static analysis described earlier.

	Checkstyle	Findbugs	PMD	JTest	SonarQube (SSLR)	Coverity	CodeSonar
Main goal	Conventions	Potential bugs	Bad practices	Metrics, analysis, runtime errors, detection and unit tests creation	Metrics, analysis and runtime errors detection	Find bugs	Runtime errors detection
Language	Java	Java	Java, JavaScript, PLSQL, C/C++, C#, PHP, ...	Java, C/C++, .NET, FDA	Java, JavaScript, COBOL, C#, Python, C/C++, Flex, ...	C, C++, Java, C# and JavaScript	C, C++
License	GNU 2.1	GNU GPL	BSD	Proprietary software	GNU GPL	Proprietary software	Proprietary software
Integration	IDE's, ANT, Maven, Jenkins, Hudson, SonarQube	IDE's, ANT, Maven, Jenkins, Hudson, SonarQube	IDE's, JDeveloper, ANT, Maven, Jenkins, JHudson, SonarQube	Integration with Parasoft's tools	SonarQube	Standalone app	C, C++ compilers (e.g. Visual Studio)
Version	6.16.1	3.0.1	5.4.1	9.6	1.21	unknown	unknown
Last release	3th March 2015	6th March 2015	4th December 2015	unknown	24th July 2015	unknown	unknown

2.2.9 Conclusion

The tools described in this section are interesting and present different ways to analyse the source code of a program. However, there are some restrictions due to the fact that Skalup already uses SonarQube. This implies, to the extent possible, to use SonarQube. As said before, SonarQube also supports some add-ons such as Checkstyle, PMD and Findbugs. The last three tools are therefore not excluded. However, the tools detailed in this chapter remains important, because there are full of ideas useful for this thesis. The different software and libraries described here will be sources of inspiration for the Design by Contract tool made for this master thesis.

CHAPTER 3

SPECIFICATION BY SKALUP

The section describes how the specifications are used among the Skalup development team to document the classes and methods. The way specifications are described is inspired from the book “Program Development in Java”, by Barbara Liskov [34].

3.1 Class and method specifications

This part focus on the specifications of classes and methods. For demonstration purpose, a class representing a segment is used. It will help understanding the different concepts introduced in the section. The class and method specifications covers specifying the behaviour of the classes and methods (what they should do) and not the way they are implemented (what they actually do). For this reason, the bodies (implementation) of methods and instance variables are not provided.

```
/**
 * This class represents the mathematical concept of a line segment.
 *
 * Specification fields:
 *   @specfield startPoint : point // The starting point of the line.
 *   @specfield endPoint : point // The ending point of the line.
 *
 * Derived specification fields:
 *   @derivedfield length : real  // length = sqrt(
 *                            (startPoint.x - endPoint.x)^2
 *                            + (startPoint.y - endPoint.y)^2)
 *                                 // The length of the line.
 *
 * Abstract Invariant:
 *   @invariant A segment's start-point must be different from its
 *              end-point.
 */
public class Segment {

    ... // Fields not shown.

    /**
```

```
     * @requires p != null && ! p.equals(this.startPoint)
     * @modifies this
     * @effects Sets this.endPoint to p
     */
    public void setEndPoint(Point p) {
        ...
    }

    ...

}
```

Figure 3.1 – The *Segment* class, example from [2]

Some important and interrelated concepts used within this chapter are defined below. The understanding of these concepts is essential for the understanding of the thesis.

Abstract Value
"What an instance of a class is supposed to represent. For example, each instance of *Segment* represents a given segment."[2]

Abstract State
"The information that defines the abstract value. For example, each abstract *Segment* has a starting point and an ending point."[2]

Specification Fields "Describe components of the abstract state of a class."[2] For instance, the abstract state of a *Segment* consists of the specification fields startPoint and endPoint.

Derived Specification Fields
"Information that can be derived from specification fields but is useful to give a name to."[2] For instance, *Segment* has the derived field length that describes the length of the segment.

Abstract Invariant
"A condition that must be true over the abstract state of all instances of a class. Abstract invariants are over a class's abstract state."[2] For example, *Segment* requires that no other segment has the same starting and end point.

Method Specifications
"Describe a method's behaviours in terms of abstract state."[2] For example, The *setEndPoint*() method in the *Segment* class reassigns the endPoint specification field.

These concepts are part of the specification of the class and are integrated in the Javadoc. Their purpose is to provide the client information on how to use the class.

3.1.1 Abstract Values and Abstract State

This part explains how the specifications are used to describe the abstract state of a class, its specification fields and derived fields. Then it explains how the behaviour of a method can be specified using the concept of pre- and postconditions.

The abstract value of an object is an abstraction of what the object represents for humans. It might be integers, circles, segments, lines, lists, strings for instance. These are high-level abstractions.

The abstract state of an object determines the abstract value of an object, it is the information that a human associates with the abstract value of an object. For instance, the string "hello world" is associated in the mind of a human by the concatenation of the following letters 'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'. The sequence of letters represents the abstract state of the string. The abstract state is not affected by the way the string is concretely constructed in the language (whether it is represented by an array, a list, or some other representations).

For some data types (classes), familiar mathematical objects or annotations can be used to represents its abstract value. For example, set of integers, sequence of characters, tuple, etc. are mathematical objects that can be used to represent in abstract value. Developers can also use mathematical annotation such as

- set union: $x \cup y$
- set membership: $a \in x$ or a in x
- sequence construction: $[a, b, c]$
- sequence concatenation: $x : y$
- sequence indexing: $x[i]$
- tuple construction: $< a, b, c >$
- etc.

However, Skalup developers are not obliged to use these annotations. The main goal of the abstract value is to describe what an object represents. This description has to be clear and unambiguous. If a developer is not at ease with these annotations or has a doubt whether other developers will understand his specification, he will define it in plain text.

Specification Fields

Most of the time, the abstract value cannot be represented by a mathematical object because the abstract value represents a more complex object made of properties. In these circumstances, the abstract value can be represented by fields. For instance, a circle has a center and a radius, a person has a firstname, a lastname and a birthdate, etc. These fields are called specification fields or abstract fields.

Most of the time, Specification fields or specfields correspond to the observer methods on the abstract data type. The specfields should be present in the overview of the class because they represent information that are primordial for any user of the class they belong to.

In the Skalup team, developpers are used to writing them as follow :
@specfield name : type // description.
The following example shows the specfields for the *Circle* class, that represents the mathematical concept of a circle.

```
/**
 * Represents a circle.
 *
 * @specfield center : Point // The center of the circle
 * @specfield radius : float  // The radius
 *
 * @invariant radius > 0
 */
class Circle {
```

Figure 3.2 – The specification of the *Circle* ADT

Specification fields have no impact on the interface or implementation of the class. Despite the fact that spec fields often correspond to observers, this is not always true. Clients might not have access to the information contained in the specification fields and therefore, there will not exist observers for the specfields, or the implementation may not have a one to one relation between its concrete fields and its specifications fields. A spec field can be derived from multiple concrete fields.

Specification fields are useful for describing the method specifications and the abstraction functions.

Derived Fields

Derived fields are fields that are derived from other specfields. They are useful if the developer want to give a name to the information they represent.

```
/**
 * Represents a circle.
 *
 * @specfield center : Point     // The center of the circle
 * @specfield radius : float  // The radius
 *
 * @derivedfield area : float // area = PI * radius^2. The area of the
 *                            circle.
 * @derivedfield diameter : float // diameter = radius * 2. The
 *                                  diameter of the circle.
 * @derivedfield circumference : float // circumference = PI *
 *                                       diameter
 *
 * @invariant radius > 0
 */
class Circle {...}
```

Figure 3.3 – The specification of the *Circle* with derived fields

The derived field *area*, from the figure 3.3, can be derived by squaring the radius specification field and multiplying the result by PI. The documentation should indicate how the specfield is derived from the non-derived spec fields.

The idea of derived fields is to simplify the specifications (method, abstract functions and representation invariants) of a class. It is indeed easier to understand when a developer is talking about the circumference of a circle rather than *PI * radius * 2*. Because derived fields are based on spec fields, they should not be described with representation fields, but only with specification fields. Moreover, it is not necessary to express the effects a method has on a derived field because the derived field is defined in terms of spec fields.

For example:

```
/**
 * Represents a circle.
 *
 * @specfield center : Point     // The center of the circle
 * @specfield radius : float  // The radius
 *
 * @derivedfield area : float // area = PI * radius^2. The area of the
 *                            circle.
 * @derivedfield diameter : float // diameter = radius * 2. The
 *                                diameter of the circle.
 * @derivedfield circumference : float // circumference = PI *
 *                                       diameter
 *
 * @invariant radius > 0
 */
public class Circle {

    private Point center;
    private float rad;

    // Abstraction Function:
    //   AF(r) = a circle, c, with c.center = r.center
    //           and c.radius = r.rad.
    //
    // Representation Invariant:
    //   rad > 0

    /**
     * @requires cent != null and r > 0
     * @effects Makes this be a new Circle c with c.center = cent and
     *          c.radius = r
     */
    public Circle(Point cent, float r) {
        Assert.notNull(cent);
        Assert.isTrue(r > 0);

        this.point = cent;
        this.rad = r;
    }

    /**
     * @return this.center
     */
    public Point getCenter() {
        return this.point;
    }

    /**
```

```
     * @return this.radius
     */
    public float getRadius() {
        return this.rad;
    }

    /**
     * @return this.circumference
     */
    public float getCircumference() {
        return this.rad * 2 * Math.PI;
    }

    /**
     * @requires r > 0
     * @modifies this
     * @effects Sets this.radius to r and keeps the center.
     */
    public void setRadius(float r) {
        Assert.isTrue(r > 0);

       this. rad = r;
    }
}
```

Figure 3.4 – The *Circle* class

As shown above, the method specification of the getCircumference() method uses the derived field $circumference$ to ease the explanation of what does the method return.

As said before, a method should not indicate its effects on derived fields because they are derived from spec fields. Similarly, the setRadius() method should not indicate what the new *area* and $circumference$ become when the *radius* is changed.

3.1.2 Method Specifications

Method specification is the description of the behaviour of the method. The description is made from preconditions and postconditions and should not refer to concrete field, but only to specification fields or to the arguments passed to the method.

Preconditions

Preconditions are properties that are required to be true when a client calls the method. The client is responsible for guaranteeing that the arguments passed to a method call respect the precondition. If it is not the case, the method is not obliged to respect the postcondition and consequently can behave in any fashion, including causing the program to crash or continuing with erroneous data. Nevertheless, it is considered as good practise to check that the input values respect the preconditions and, if they don't throw a descriptive. This is called "defensive programming". However, clients should never assume that the preconditions are checked and they should always guarantee that the preconditions are respected.

The preconditions are described with the *requires* clause in method specifications. The *requires* clause is optional and if the clause is left out, the method is implicitly assumed to have a true precondition.

Developers at Skalup uses defensive programming by checking these preconditions programmatically with home made assertions. A class gathering the most useful assertions was created among which:

- notNull(...) : asserts that the given variable is not null
- isNull(...) : asserts that the given variable is null
- notEmpty(...) : asserts that the given collection is not null and not empty or that the given string is not null and not empty
- isTrue(...) : asserts that the given boolean expression is evaluated to true
- isFalse(...) : asserts that the given boolean expression is evaluated to false
- shouldNeverGetHere() : assert that this part of the code is never reached
- equals(...,...) : asserts that the two given parameters are equals

The use of assertions is shown in the following figure 3.5, where the assertion checks that the variable r if greater than 0. They are used in the same way as Junit assertions (e.g. *assertEquals(Object expected, Object actual)* or *assertTrue(boolean condition)*), but their goal is different. Junit assertions are meant to make a unit test succeed or failed, however the assertions used here, are meant to make the execution of a method fail if the precondition is not respected. Moreover, Junit assertions are executed only when the tests are run (most of the time during the compilation). Meanwhile, the assertions used to verify the precondition are executed at run time, each time the method is called.

```
...

/**
 * @requires r > 0
 * @modifies this
 * @effects Sets this.radius to r and keeps the center.
 */
public void setRadius(int r) {
    Assert.isTrue(r > 0);

    rad = r;
}

...
```

Figure 3.5 – Example of usage of assertions

requires (default: no constraints)
The preconditions that must be met by the method's caller.[2]

Postconditions

Postconditions (in contrast with preconditions) are properties that are guaranteed by the class' developer to be true when a method exits. Nevertheless, if the preconditions are violated by the client when he calls the method, the postconditions are not required to be true. As a consequence and as said before, the method can behave in any fashion.

A post condition can be written as a single plain sentence but it is more convenient to split it into separate parts. The Liskov book [34] uses $modifies$ and $effects$; the Skalup developers use $modifies$, $effects$, $return$, and $throws$.

Below, the "default" keyword means what is assumed if the clause it is associated with is left out.

modifies (default: nothing, which means that there are no side effects)[2]
Specifies the variables that may be modified by the method.
The variables may or may not be modified, nothing obliges the listed variables to be modified but it is a possibility. If an object a has spec fields m, n and o, then the clause *@modifies* a means that any of $a.m$, $a.n$ or $a.o$ variables may be modified. Developers can be more restrictive if they specify a particular spec field in the $modifies$ clause. For example: *@modifies* $a.n$, means that

only *a.n* may be modified. The variables that are not listed in the modifies clause are guaranteed to be unchanged.

effects (default: true, which means "can have any effect" on the references listed under modifies)[2].
The "effects" clause defines the behaviour of the procedure, the side effects and the changes to the state of the current object. However, the effects clause does not indicate what happens if the preconditions are not respected.

In contrast with the modifies clause that indicates what variables can change, the effect clause indicates what changes are made.

return (default: no constraint on what is returned) [2] Specifies what value is returned by the method, if any.

throws (default: none, which means that no exceptions are ever thrown)[2] The exceptions that might be thrown during the execution of the procedure. Theses exceptions are thrown when the procedure matches a specific condition defined by the postcondition that requires an exception to be thrown.
e.g. *@throws NotPossibleException if ref does not reference any product in this.*

Using Spec Fields for Specifications

Spec fields are used to write the specifications of the ADT's operations. Here is a example of a specification for the Circle class:

```
/**
 * @requires r > 0
 * @modifies this
 * @effects Sets this.radius to r and keeps the center.
 */
public void setRadius(float r) {
    ...
 }
```

Figure 3.6 – Example of usage of spec fields for specifications

The specifications cannot refer to concrete fields (e.g. rad), instead they should refer to specification fields such as *radius*. The concrete fields or representations fields can be implemented in many different ways (e.g: the radius can be expressed by a float, a double or Double, a String, of even a Radius object) and so, the specifications should not divulge the concrete fields.

Using Derived Spec Fields for Specifications

Derived fields are shorthands for writing specifications because derived field can be written in terms of specification fields. For this reason, derived field can be used in a method specification.

```
/**
 * @return true if this.diameter > diameter.
 */
public boolean larger(float diameter);
```

Figure 3.7 – Example of usage of derived fields for specifications

Subclasses and overridden methods

A subclass must have a stronger specification than the superclass it inherits. Its abstract state may also be larger than the one from its superclass due to added class members. When the specifications are equivalent in the superclass and in the subclass, there is no need to duplicate them in the subclass. However, it is convenient to add a brief note announcing that the superclass documentation should be used as a substitute. This can be useful to determine whether the specification is the same, or the developer inadvertently omitted to document the class.

When the specifications diverge from the superclass' specification, developers have the choice between two options. First, they complement the existing specifications which leads to distribution of the specifications in the class and in its parent's class. The second approach is to copy the specifications from the superclass in the subclass and complete it with the characteristics of the subclass. This approach has the advantage that the entire specification is in one place.

Whichever approach is chosen, the developer has to clearly express his choice, and that he is consistent everywhere in his codebase.

The same rule applies when a method overrides another method. It is tolerable to omit the specifications on an overridden method if the behaviour is the same as the corresponding method in the superclass.

3.2 Abstraction Functions and Representation Invariants

This part describes how a class' implementation is documented by Skalup developers.

As an example, the same class as the previous one, representing a Segment, will be used. This time around, the implementation is now shown because this part is about detailing a class's implementation as opposed to its specification.

```
/**
 * This class represents the mathematical concept of a mutable line
 * segment.
 *
 * Specification fields:
 *
 * @specfield startPoint : point // The starting point of the segment.
 * @specfield endPoint : point // The ending point of the segment.
 *
 * Derived specification fields:
 *
 * @derivedfield length : real // length = sqrt(
 *                              (startPoint.x - endPoint.x)^2
 *                              + (startPoint.y - endPoint.y)^2)
 *                              // The length of the line.
 *
 * Abstract Invariant:
 * @invariant A segment's startPoint must be different from its
 *           endPoint.
 */
public class Segment {

    private int startX;
    private int startY;
    private double length;
    private double angle;

    // Abstraction Function:
    //  AF(r) = a segment, s, such that
    //     startPoint = (startX, startY)
    //     endPoint.x = startX + length * cos(angle)
    //     endPoint.y = startY + length * sin(angle)
    //
    // Representation Invariant:
    //  ! (startX == startX + length * cos(angle)
    //        && startY == startY + length * sin(angle))

    /**
```

```
     * @requires startX >= 0, startY >=0, length > 0
     *          and 0 <= angle < 360
     * @effects Makes this be a new Segment s with
     *         s.startPoint = (startX, startY)
     *         and s.endPoint = (startX + length * cos(angle),
     *                           startY + length * sin(angle))
     */
    public Segment(int startX, int startY, double length, double
        angle) {
        Assert.isTrue(startX >= 0);
        Assert.isTrue(startY >= 0);
        Assert.isTrue(length > 0);
        Assert.isTrue(angle >= 0);
        Assert.isTrue(angle < 360);

        this.startX = startX;
        this.startY = startY;
        this.length = length;
        this.angle = angle;
    }

    ...
}
```

Figure 3.8 – *Segment* class with its implementation

Concrete Representation

"How the abstract state of a class is represented within a Java object."[35] For instance, *Segment* uses a int for its concrete field *startX*;.

Abstraction Function

"A function from an object's concrete representation to the abstract value it represents."[35] The abstraction function for *Segment* can be expressed as follow:

```
    // Abstraction Function:
    //  AF(r) = a segment, s, such that
    //     startPoint = (startX, startY)
    //     endPoint.x = startX + length * cos(angle)
    //     endPoint.y = startY + length * sin(angle)
    //
```

Figure 3.9 – Abstract function for the *Segment* class

Representation Invariant "A condition that must be true over all valid concrete representations of a class. The representation invariant also defines the domain of the abstraction function."[35] For instance, *Segment* requires that the start point and the end point are not equals.

Abstraction functions and representation invariants are for internal use only. They document the implementation of the class and consequently should not be exposed to clients because they do not need to know these pieces of information to properly make use of the class. The Abstraction functions and representation invariants should not appear in the Javadoc, so developer makes usage of simple comments to represent them ("//" or "/*" and "*/").

3.2.1 Abstraction and concrete representation

This section details how to bind the concrete representation of the class to its abstraction, through the use of Abstraction Function and Representation Invariant.

An abstraction function AF binds the concrete representation of an ADT (abstract data type) to the abstract value represented by the ADT.
More formally:

$$AF : R \rightarrow A$$

where R is the set of rep (representation) values, and *A is the set of abstract values.* [35]

For instance, for the following ADT,

```
final class Complex {
    private final double real;
    private final double imag;
}
```

Figure 3.10 – *Complex* ADT

R is the set of Complex objects, and A is the set of Complex numbers.

Definition of R and A

The R is defined by the fields present in the class. A, however cannot be represented in the source code of the class, but both the client and the developer want to know what abstract type does the class represent. For this reason, the A, representing the abstract value space is detailed in the class overview.

```
/**
 * Complex represents an immutable complex number. <=== this is A
 */
public final class Complex {
    private final double real; <=== these fields form R
    private final double imag;
    ...
}
```

Figure 3.11 – R and A

The mutability of the class should also be specified in the class overview. Knowing whether the class is mutable or immutable is an important information.

3.2.2 Abstraction function

An abstraction function is a function. For this reason the conventional notation AF(r) is used.

```
/**
 * Complex represents an immutable complex number.
 */
public final class Complex {
    private final double real;
    private final double imag;

    // Abstraction Function
    //   AF(r) = r.real + i * r.imag
    ...
}
```

Figure 3.12 – Abstraction Function

The r of the AF(r) function represents an element in the representation space. To put it differently, the right side of the abstraction function can refer to the fields of the object r.

r symbolizes the representation. Liskov uses the letter c [36] instead, but c and r represents the same thing.

For readability purpose, it is tolerated to omit the left-hand side of the abstraction function and r is assumed to represent *this*. This shorthand makes it easier to read and understand the AF.

The following example shows the abstraction function with the left part of the function being left out.

```
/**
 * Complex represents an immutable complex number.
 */
public final class Complex {
    private final double real;
    private final double imag;

    // Abstraction Function
    //   real + i * imag
    ...
}
```

Figure 3.13 – Abstraction Function bis

Sometimes, ADTs have one-to-one correspondence between the specfields and representation fields. The following example illustrates this particular case.

```
public class Segment {
    private Point start;
    private Point end;

    // Abstraction Function
    //   AF(r) = line l such that
    //     l.start = r.start
    //     l.end = r.end
    ...
}
```

Figure 3.14 – AF one-to-one correspondence

The correspondance between the abstraction and the representation is trivial, so the AF is not needed in this case.

3.2.3 Representation Invariants

A rep invariant (RI) binds the concrete representation to a boolean value (true of false).
More formally,

$$RI : R \rightarrow boolean$$

"where R is the set of rep values. The rep invariant describes whether a rep value is a well-formed instance of the type."[35]

```
// Rep Invariant
//   RI(r) = r.name != null && r.balance >= 0
```

Figure 3.15 – RI example 1

For readability purpose, the RI(r) is often omited and the developers write the representation invariant as a predicate that must be true.

```
// Rep Invariant
//   name != null && balance >= 0
```

Figure 3.16 – RI example 2

A representation invariant often uses Java syntax with abstract math syntax or sometimes it is a simple description made of plain text. The last approach is permitted as long as the representation invariant is not ambiguous.

For example:

```
//    for all i in 0..n-1, transactions[i] instanceof Trans
```

Figure 3.17 – RI example 3

```
//    suit in {Clubs,Diamonds,Hearts,Spades}
```

Figure 3.18 – RI example 4

Developers can refer to other ADTs either by their spec fields or by their pure methods.

For instance, the Transaction type has a specification field amount that is reachable by the method *getAmount*. Then the rep invariant for the Account class (which makes usage of Transaction objects) might be represented as follow:

```
(1)  //   balance == sum (i in 0..n-1) of transactions[i].amount
```

Or

```
(2)  //   balance == sum (i) of transactions[i].getAmount()
```

Or even this

```
(3)  //   balance = sum (i) of transactions.get(i).getAmount()
```

These different notations are equivalent and it is up to the developer to choose which one he prefers.

It is most likely that developers will be the readers of the specifications, AF and rep invariants, not a program. It might be programmers trying to debug, update the code, taking over the code after a developer has left the team or even the developer who wrote the specifications in the past who is now trying to remember how the code works.

For these different reasons, simplicity and clarity are the most important criteria (more important than syntactic correctness) when writing the rep invariant. That does not mean that developers should renounce to semantic precision and avoiding ambiguities, but they should think twice before writing complex things like this:

```
// Abstraction function is
//   <[x,y],s^[for all 0<i<=chars.size(), chars[chars.size()-i]]> |
    x=front.cdr,
//    y=back.car, s=n.toString()>
```

Figure 3.19 – Complex example of a Abstraction Function

3.3 Semi-formal specifications

The Semi-formal specifications take the best from both worlds; informal and formal specifications.

Formal specifications are based on the semantics of programs, in other words, they are based on formal mathematical descriptions of the meaning of a given program source code. They are used in the development of the most critical software (e.g. control of nuclear power plants).

Informal specification are not based on mathematical descriptions. They offer more freedom to developers in order to describe what their programs do.

Formal and semi-formal approaches have their advantages and drawbacks: The informal specifications are easier to understand and to define. However, they can be ambiguous. On the other hand, the formal specifications are more difficult to learn and understand for person that are not familiar with such specifications. However they provide mathematical rigour and remove all ambiguities.

The semi-formal specifications are based on both worlds. They are more precise than informal and easier to understand than formal specifications. They offer freedom to developers; if they need to describe the behaviour of their code without ambiguities, they are free to use mathematical annotations. However, it may be more difficult for some developers to understand those annotations.

Semi-formal specifications have also drawbacks; they are not as precise as the formal approach and not as easy to write and understand as the informal specifications. But their goal is to find a balance between these worlds, and to be easily understandable by developers without being ambiguous.

CHAPTER 4

RESEARCH AND CONTRIBUTION

This chapter explains the work accomplished during the realisation of this master thesis, it consists of the realisation of a tool to suppport Design by Contracts using semi-formal specifications in software source code. First, the section 4.1 focus on the different checks that can be performed on the specifications to ensure that they are valid. This section lists the tests and explains their behaviour. The section 4.2 concentrates on the tests that where implemented and explains the prototype realised that materialises the researches conducted for this master thesis.

4.1 Research on potential static verifications

As aforementioned, this section will detail verifications that can be executed on the documentation of a program that uses the specifications explained in the previous chapter.

It is interesting to add tests to ensure that the specifications are met by the implementation of the developers and that the source code does not differ from them as the program evolves. Furthermore, these tests will make code reviews more accurate and more effective. The different verifications are explained hereafter.

4.1.1 Class and method specifications

There are several tests that can be performed to ensure that the specifications documented in the overview of a class are correctly used among the other specifications of the class. This sub section contains the tests related to the class and method specifications, they are grouped by the following categories:

— Usage of specification fields and derived fields
— The mutability of a class
— Constructors
— Method specifications

Usage of specification fields and derived fields

1) Developers can refer to spec fields defined in the overview of a class by using *this.* followed by the name of the field (e.g. *this.radius*)

However, it is possible that the developer refers to an erroneous field. It might have been removed, it might never have existed or the developer might have misspelled it. With this in mind, we can imagine a test that checks for every reference to a specfield, for instance "this.radius", and ensures that they are correct and so that the corresponding specfield or derived field does exist in the corresponding class. Nevertheless, a reference to a field that is not defined in a class is not always incorrect. Indeed, in the case of a class that has a superclass, the subclass inherits from its parent's spec fields. But as explained in the previous chapter, the developer can choose either to copy them in the subclass or omit them. In both case, the test must be able to check the reference of a spec field against its superclasses. This process is recursive, and so we can imagine that a class refers to a specfield defined in the superclass of its superclass.

There may be plenty of references to spec fields in a class. These references to spec/derived fields can be found in the definitions of the derived fields, in the abstract invariant definition, in the specification of the constructors or in the contracts of the methods. Every time a reference to a spec field is made, the check has to ensure that the field is defined, either in the class itself, or in one of its parents.

If the verification determines that one spec field or derived field is not defined or is defined multiple times in the same class, the test should log the error to warn the developer that a mistake has crept into its source code.

The goal of this check is to ascertain that the references to spec fields are correct, and that the specification stays up to date. When renaming a variable, refactoring tools ensure that all occurrences of the variable are renamed. Similarly to variable renaming, this test will guarantee that changing the name of a spec field will inform developers when they forget to rename an occurrence.

2) Inner classes represent a particular case where a class (the inner class) has access to the specfields of another class (the outer class) by using the *this* keyword. As an illustration, the class B is an inner class in the class A, then, B can access the specification fields of A, but A does not have access to the spec fields of B by using *this.* followed by the name of the field. The test should support this particular case and ensure that the outer class does not reference a field of its inner class by using the *this* keyword.

3) Another feature that could be supported is chaining calls to specification

fields. For instance, it would be confortable for developers if a test could ensure that complex references to specfields such as *this.person.city.name* was supported. The test would verify that the *person* specfield is defined for the current ADT, then verify that the *city* specfield is defined on the ADT represented by *person* and finally checks that the specfield *name* exists on the ADT *city*.

The mutability of a class

4) When a programmer specifies an ADT, it is recommended that he precises the mutability of the abstract data type. If the ADT being specified is defined as immutable (which means that the visible state of the object cannot change), it might be interesting to perform tests on the ADT itself to assure that the methods will not change its state. This verification can be done by analysing the specifications of the methods provided on the ADT. These methods should naturally not contain the clause @$modifies$ or @$effects$.
On the contrary, if the ADT is defined as mutable, it would be a good idea to check that there exists at least one method capable of modifying the state of the ADT. If there is no such method, then the verification could tell the developer that he can define its class as an immutable object. This information can also be easily checked by verifying the mutability of every method or non-private attributes. Currently the mutability of the class is defined in the overview of the class as plain text (e.g. Circle represents a mutable ...) as shown in chapter 3 (page 28). Introducing annotations to define the mutability of the ADT, such as @$Immutable$ and @$Mutable$ could ease the detection of such information and highlight the mutability of the ADT.

5) Moreover, it could be interesting to ensure that calls made in the implementation of the ADT may not mutate the state of the ADT. Calls to mutator methods in the implementation can indeed change the state of the object, which is in contradiction with the specification of the ADT. These mutator methods can be recognized by their specification if any (methods with @$modifies$ clauses). It is also possible to try to recognise mutator methods that do not have specification. The most common mutator methods defined by Java can be listed or guessed. These methods generally start with keywords such as *set*, *remove*, *add*, *update* or *delete* and may implicitly be considered as mutator methods. The check can verify that the implementation does not call any of these methods.

For instance, if an immutable class has methods in which a method having an @$modifies$ annotation is called, the test will log an error to inform developers

that suspicious call to a mutator method was detected. Another example could be illustrated by an immutable ADT that uses a *Collection* in its representation. If the implementation calls methods such as *remove*(...) or *add*(...) on the *Collection* class, the test can guess that these methods are mutable. Because the ADT was previously set as immutable, this behaviour is forbidden, and the developers should be informed that an error was detected.

6) Moreover, if the implementation of an immutable object uses mutable object, such as *List*, *Array*, *Date*, the developer should avoid exposing the representation. If the representation is exposed, an immutable object could be mutated by modifyng a mutable object of its representation. In that case, it is recommended that the developer returns a copy of the mutable object used in the representation in order to avoid any external modification of the state of the ADT. This can also be done by encapsulating the mutable object into a read-only object, in the same way as $Collections.unmodifiableList(...)$ does [37].

The representation can be exposed in several ways. First, it can be exposed through the constructor. If a mutable object is used as a parameter of the constructor and if the representation references directly the given object, the methods that call the constructor of the object has also the reference to this mutable object. Therefore, the caller can mutate the immutable object by modifying the mutable object passed through the constructor. This can be avoided by "copying" the object or encapsulating it in a read-only object. Similarly, the problem also applies to the methods with parameters.

Then, the representation can also be exposed through the return value of a method. If a method of an immutable class returns a mutable object of its representation, then, the caller of this method will be able to modify this object, which is in conflict with the ADT specification. For example, a *Classroom* class, that is immutable and has a list of *Pupil*, provides an observer method to get the list of its pupils;

```
...
/**
 * @return this.pupils
 */
public List<Pupil> getPupils() {
    return this.pupils;
}
...
```

Figure 4.1 – Exposition of the representation

The code above exposes the representation, because anyone calling this method

will be able to add or remove elements in the list. A way to correct this method could be to return an immutable collection (supposing that *Pupil* is immutable);

```
...
/**
 * @return this.pupils
 */
public List<Pupil> getPupils() {
    return Collection.unmodifiableList(this.pupils);
}
...
```

Figure 4.2 – Correction of the exposition of the representation

The test should check for representation expositions and log a message to inform developers if it happens.

7) Furthermore, immutable objects should override the *hashcode* [38], an *equals* [39] methods. The *equals* method must be used to represent the behavioural equivalence of two objects. Both objects should be equals if no sequence of access to their methods can distinguish them. A mutable object is only equals with itself, i.e. *equals* and == must provide the same result. Two immutable objects are equals if they have the same state[40], therefore an immutable object must override the equals method because the default implementation (which compares the references of the objects) is not suitable for immutable objects[36]. If a class overrides *equals*, it must also override the *hashCode* method. The reason is if two objects are equal, then their *hashCode* method must return the same values. A verification can easily be made to ensure that both methods are overriden in immutable classes.

The implementation of these methods should include all the class fields that are present in the abstraction function. This can also be verified, because sometimes developers add information on the class and forget to update the *equals*()[39] and *hashCode*()[38] method.

Constructors

8) In Java, constructors are particular methods that create an instance of a class, called an object. Constructors should be specified, indicating how the object is being constructed. These specifications are described in terms of preconditions and postconditions as mentioned in the previous chapter. The post condition should indicates for all spec fields the value they are being initialized to. It means that every single spec field defined for the ADT (including

the spec fields defined in a super class) should be referenced in the @*effects* clause of the constructor. Additionally, the @*effects* clause is mandatory for the constructors and should always be present, even if there is no spec field defined for the ADT. Indeed, this clause indicates what the object is being initialized to and without this information, clients wouldn't be able to use the ADT correctly. However, the @*modifies* clause is omitted on the constructors. The explanation of this omission is that at the time the constructor is being called, the object does not exist yet and so, it cannot be referenced by the *this* keyword, because *this* has not been instantiated yet. However, if the constructor has side effects (on parameters), the modifies clause should exist on the constructor, indicating what are the side effects. Nevertheless, such a behaviour may indicate bad practise or bad design.

A test could verify that the @*effects* clause is complete and does not forget any spec fields in its description. Complete means that every specfields defined for the class should appear in this clause. Fields that are not being initialized to a value passed as a parameter of the constructor should also tell the value they are being initialized to. For instance, they can be initialized to *nil*, to an empty list (e.g. {}) or any other default value.

9) Another verification that could be interesting would be to verify that preconditions are defensively checked by assertions. As said in the specification chapter, defensive programming is a good practice when dealing with preconditions. However, it may sometimes be difficult to check those assertions (e.g. if they are not evaluable as a boolean expression) and sometimes, assertions are omitted for performance purposes (e.g. an assertion that checks that a list is ordered in a method that is called many times may have an impact on the execution). Making the assertions parametrizable would prevent developers to remove them when their execution has a impact on the application. Instead developers would parametrize them to disable them when the software if deployed in production.

10) Over time, some specifications became standard within the Skalup development team. For instance, the description of the @*effects* clause on constructors of an ADT if often written in the following way:

Makes this be a new ... where ...

Where the first gap is filled with the type of the ADT and the second one is filled by the description of the initialisation of the different spec fields.

For instance, the @*effects* clause for the *Circle* class, used in the previous chapter would be

"Makes this be a new **Circle c** with **c.center = cent and c.radius = r**"

Where **Circle** is the type of the ADT, **c.center** refers to the spec field *center* and **c.radius** refers to the spec field *radius*.

A check could be established to standardise the description of the @$effects$ clause on ADT, but it would require that some developers change their habits. Or the test could simply suggest templates that could help developers without, however, being mandatory.

Method specifications

The specifications on methods are quite similar than the specifications of the constructors.

11) The preconditions should also be defensively verified on public methods and so, a check could be responsible for ensuring that assertions are verifying input values.

12) An analysis could determine if an instance variable (representation field) that is present in the abstraction function is modified and if this is the case, check if the @$modifies$ tag is defined on this method. Similarly, the analysis could detect when a mutator method is called and check for the existence of the @$modifies$ tag as explained in the section describing the verifications on mutable and immutable class.

In the same fashion, the @$effects$ and the @$modifies$ clauses are tightly coupled, except for the constructors, and so, they should always coexist (the omission of the clause @$modifies$ on constructors is an exception). A simple test can easily verify this coexistence.

It would also be a good idea to check that the @$return$ and @$throw$ clauses are not omitted. These tags indicates when the methods return a value or may be susceptible to throw an exception. Some IDE already check for these omissions.

13) It is also important to notice that all parameters of a method should appear in the post condition. Indeed, if they do not appear in the post condition they can be considered as useless, due to the fact that the postcondition describes what the method does, so if a param does not appear in it, it is not used by the method. A check could inspect if each param appears at least once in the postcondition, namely, in the @$modifies$, @$effects$, @$return$ or @$throws$ clauses. If a param or a specfield appears in the @$modifies$, it also needs to be in the @$effects$ clause.

14) In addition, params may be from an ADT on which specfields are defined. These specfields can be referenced by using the name of the param, in the same

way a specfield is referenced using "this.". A test could verify that these references on params are used accurately and that the specfield being referenced does exist in the ADT being represented by the param. As explained before in this section, chaining specfields is a useful functionality. This functionality should also be supported for the params of methods and constructors, so a test should be able to identify undefined or erroneous specfields used while chaining references to specification field on the params of a method.

15) Finally, it could be interesting to introduce the *old* keyword, even if it is almost never used by Skalup developers, it could facilitate specifications when willing to reference the value of a field (specfield or param) on entry of the method. The old keywork can only be used in the postcondition. In the same way as Cofoja, another extension can be added; the *result* keyword, it is used to reference the value that will be returned by the method. It could be used to write the postcondition (e.g. result >= 0). A test could verify that the usage of these keywords is appropriate and that none of the previous rules is transgressed. Knowing that there exist verification for these keywords Skalup developers may use them in the future.

4.1.2 Abstraction Functions and Representation Invariants

It might also be possible that the abstraction function or that the invariant representation are nonconform with the concrete implementation of the class with which they are associated. In order to remedy this problem, we can imagine verifications, which checks the concordance between these elements.

The following tests are categorized as follow:
- Abstraction Functions
- Representation Invariants

Abstraction Functions

16) The abstraction function is not mandatory in an ADT, but recommended. If the choice of the developer was to define such a function, we can ensure that the specfield used in that function are defined correctly, and that the concrete fields they are mapped to, do also exist. If a field is deleted or renamed, the developer may forget to update the abstraction function. The test will prevent the abstraction function to reference erroneous specification or representation fields. The test could also guarantee that all defined specification fields are present in the abstraction function and that no specfield have been left out. This error may appear when developers add new specfields on their ADT but forget to update the abstraction function. It may be interesting to remind

that the representation fields are prefixed by 'r' that is a variable that stands for a concrete instance. This test could go further by ascertaining that the types defined in the specifications and in the representation are compatible. Indeed, a specfield could be typed as a primitive int, but its representation can use a different type, such as Integer, long, etc. Moreover, there are several ways to express some data structures such as lists. For example, the type list is compatible with collections, arrays, a sequence of elements etc. This is a boundary case, but is nevertheless often used by developers. This test will not guarantee that the abstraction function will be error-free, but it will detect the most common errors in AF.

17) It is also possible to perform a simple verification ensuring that the developer used the correct sort of comments to describe its abstraction function. As said before, the AF is for internal use only and so, the comments used should not be Javadoc comments (Javadoc comments will be exported to the Javadoc when the documentation of the program will be generated, and so will be publicly accessible). The programmer can use the following comments to express the AF of the ADT; '//' or '/*' and '*/'. Currently, developers defines the AF by starting with a plain string indicating the begining of the AF.

e.g. *// Abstraction Function*

But to ease the detection of such functions, developer could use a tag such as *@AF* or *@AbstractionFunction*.

This verification will guarantee that the abstraction function will not be published in the public API of the program.

Representation Invariants

It is good practice for developers in the Skalup team to define a method called *checkRep*() that asserts that the invariant is respected within the object. The *checkRep* method should be called every times the internal state of the object has been modified, either at the begining or at the end of a public method. The *checkRep* method is made of assertions that assert that the state of the object is valid.

Given these points, three tests can be imagined.

18) First, a test that checks for the presence of a representation invariant. If so, it checks if a *checkRep*() method is defined. In the case the developer forgets to write the *checkRep* method, but has defined a representation invariant, he should be warned that the method is missing.

19) Secondly, we can test that the *checkrep* method is called when it should be. As said before, the representation invariant should be respected within the object at entry and exit of public methods. So developers should check that the invariant is respected before or after the object is modified. As a mean to do this, the check should verify that the method is called at the end of the constructors and at the beginning and at the end of every mutator methods (such as setters, add & remove methods, etc.), or more generally, each method that has a *@modifies* tag in its specification.

20) It may also be possible to extract information from the definition of the representation invariant and verify that the *checkrep* method performs the appropriate tests.

```
// Rep Invariant
//   RI(r) = r.name != null && r.balance >= 0
```

Figure 4.3 – Example of representation invariant

For instance, we could easily extract information from the previous snippet 4.3, such as:

— r.name should not be null
— r.balance should be greater or equal to zero

With these pieces of information, the test can verify that two assertions are defined, one for each statement.

However, it would be more challenging to detect correct and relevant information when facing more complex semi-formal representation invariants such as:

```
/*
 * Representation Invariant:
 *   for all <k,v> in attributes + children,
 *     k != null && v != null && k = v.name
 */
```

Figure 4.4 – Example of more complex representation invariant

4.1.3 Summary

This sections summarizes the different tests explained above and details the methodology used to define these tests.

Discussions with the development team of Skalup permitted to define their needs and what are the most common mistakes made by the developers. Moreover, an analysis of the way they use specifications allowed to define verifications that could detect inaccuracies. Depending on the feasibility and their interest, some test have been prioritized in order to be implemented within a first prototype. A first analysis was made to discuss the criteria to define the different tests and a second one was about validating the different tests to be implemented for the prototype realized in this thesis.

The following table summarizes the different tests detailed above.

Table 4.1 – Summary of the different tests

Category	Test	Summary
Usage of specification fields and derived fields	1	Existence of specfields and derived-fields
	2	Access of specfields on inner classes
	3	Chaining specfields
The mutability of a class	4	Mutability of methods in immutable classes
	5	Call to mutator methods in immutable class
	6	equals & hashcode
	7	Representation exposure
Constructor	8	Postcondtions of constructors (@effects & @modifies)
	9	Defensive checks of preconditions
	10	Suggestion of templates for specifications on constructors of ADT
Method specifications	11	Defensive checks of preconditions (same as the test 9)
	12	Omission of specification tags (e.g. @modifies, @effects, @throws, etc.)
	13	Params appears in the post condition (@modifies, @effects, @return and @throw tags)
	14	Chaining params (similar as the test 3)
	15	Old & Result keywords
Abstraction Functions	16	Presence of the specfields and the representation fields in the AF
	17	Type of comments for the AF definition
Representation Invariants	18	Coexistence of the Representation Invariant and the checkRep() method
	19	Call to the checkRep() methods when needed
	20	Extraction of information from the Representation Invariant

4.2 Prototype of the specifications analysis tool

As highlighted in the introduction of this chapter, the following section will detail the prototype made during this master thesis. The following points will be addressed; the reasons that led to choose this technology, the problems encountered, the explanation of the approach and how the API works and finally the different tests that have been implemented.

4.2.1 The technology

According to the analysis of the different tools in the *Background and Related Work*, SonarQube seems to be the most appropriate static analysis tool to meet the requirements for this master thesis.

First of all, SonarQube is a standalone application and is ready to use. So there is no need to develop an entire toolbox that will take responsibility for executing the different tests, that will compute the different metrics during the analysis and that will keep track of these metrics to compare them over time.

Secondly, SonarQube provides a rich and user friendly graphical interface that puts forward the different metrics allowing the development team to easily get important information about the code quality. This information can then be used by developers to improve the quality of their code, but also by the project manager to ensure that the quality goals are reached by his team.

Third, SonarQube, in addition to having its own analysis engine, supports natively other static analysis tools such as Checkstyle, PMD and FindBugs. A very important point is that the predefined rules can be extended and other rules can also be incorporated in the tool. This is a crucial point because the custom verifications on the specifications of classes and methods have to be executed by the tool to meet the requirements. Moreover, SonarQube seems to be a wise choice because it provides multiple functionalities and different ways to broaden its processing capacity. Additionally, SonarQube is broadly used in the software development industry and therefore has no longer needs to prove itself as a code quality improvement tool.

And last but not least, the choice of SonarQube was strongly influenced by the fact that this tool is already used in the Skalup development team. A constraint of this thesis was therefore to use to the extent possible SonarQube. Furthermore, SonarQube supports the Java language as well as C#. These two languages are the main programming languages used to build programs among the team of developers. In first place, the prototype developed in this master

thesis will only focus on the analysis of the specifications in Java programs but may be extended to C# programs in the future. For this reason, the support of C# in addition to the support of Java is a significant point.

To summarize, the choice of using SonarQube is dictated by the fact that the Skalup team uses it on a daily basis. Nevertheless, SonarQube has good qualities, which confirms that using this tool for the development of the prototype is a wise choice. However, it is important to realize that choosing SonarQube as a static analysis tools lets the freedom of choosing the scanning engine that will analyse the Java files, such as CheckStyle, PMD, Findbugs, SSLR or a new plugin.

4.2.2 Choice of the analyze engine

The first attempt to realize a prototype to verify specifications, was done with SonarQube API. The API is known as SSLR for SonarSource Language Recognizer. This API provides methods to visit an AST made of nodes that represents the content of the Java file being analysed. However, this API does not provide methods to analyse the comments of a Java class. It is however possible to analyse them but not with the AST visit explained before, because the nodes containing the comments are ignored during the visit. The lack of functions to analyse comments easily is problematic as the prototype need to be able to read and parse the specifications being expressed as comments in the Java code.

This API also suffers from a lack of documentation and its community is almost non-existent. This would help developers willing to make custom checks for their code, as this thesis is trying to achieve.

The SSLR API was used to realise a first prototype. It was able to parse the specifications and extract the spec fields from these specifications. However, this API does not provide functionalities to analyse each file of the project and log messages later. It is only possible to log messages during the visit of a file. This functionality is however required for the prototype to be able to detect errors correctly, such as undefined specfields. Moreover, the prototype could not run with SonarQube because SonarQube was unable to resolve the correct dependencies of the projects and there were conflicts between the version of the instance of SonarQube that was running and the version used in the plugin. The lack of documentation and the poor community did not help to solve this problem.

Because SSLR has some lacks in its API, the API from Checkstyle was considered. Moreover, plenty of examples are available online to help understanding the way Checkstyle works. CheckStyle provides two way to scan files, each

one with pros and cons. The first one provides a similar API than the one from SSLR, with a visit of an AST made of nodes but it is not possible either to log error messages after all the files of a project have been analysed.

The second API however, provides such a functionality. It permits to log messages once the whole project has been analysed. This allow the check to scan all files of a project. During this process, it will collect all the relevant information it needs, such as the specification fields of the different classes. After all files are analysed, the API allows to come back on each file and log errors. This will allow the test to check specfields in superclasses of a class, because all files have been parsed (and thus all specfields are known) before verifying that the specfields are defined. However, the drawback of this API is that it does not provide such a convenient way to scan a file as the two previous API did. This API gives direct access to the lines of the file as a list, and developers have to parse the lines manually. So there is no AST visit mechanism to help developer navigating in the classes.

4.2.3 The Application Programming Interface

As explained previously, Checkstyle is a toolbox that provides facilities to developer to help them following the coding conventions. Moreover, it can also be extended with custom checks that will test the code to ensure that theses conventions are respected. This section will focus on the explanation on how the extension of the tool is possible, the limitations of the tool and the details of the API provided by CheckStyle.

In order to realise a test, two different approaches exist. Each one depends on what the developer wants to test.

The first approach is a classic visit of the nodes of an AST (Abstract Syntax Tree). The API provides a *Check* class in the package com.puppycrawl.tools.checkstyle.api. This class has to be extended by the custom class used for the realisation of the test and the *Check* class provides several methods. The main methods are explained below.

void beginTree(DetailAST rootAST) This method, as its name says, is called at the beginning of the visit of the tree. The given argument *rootAST* is the root node of the AST of the Java file being analysed. An object of type *DetailAST* contains information such as references to its parents, to its children, its siblings, but also the line number at which the node as been declared in the file and so on. All the information are important while analysing a node. They allow the developer to navigate in the tree to collect other information (e.g. when on a method declaration node, the developer can visit the children of the node to get information about the arguments of the method).

void finishTree(DetailAST rootAST) This method, in contrary with the *begintree* method, is called after the tree is visited. This allow the developer to perform some operations after each tree/file has been visited.

abstract int[] getDefaultTokens() This method is abstract and has to be overridden in the test class. The method is supposed to return the list of all token the developer is interested in. That applies a sort of filter when visiting nodes, as the visit only stops on nodes having their type in the list.

void visitToken(DetailAST ast) This method is called when the visit of the AST encountered a node of a type included in the list returned by the *getDefaultTokens* method. This method is the most useful to perform the analysis of a given file.

void leaveToken(DetailAST ast) This method is quite similar to the previous one, but is called after the node has been visited.

void log(int lineNo, int colNo, String key, Object... args) and **void log(int line, String key, Object... args)** allow the developer to log a message or a warning. This message/warning will then be displayed to the user, giving him information on what is wrong with the source code. The only difference between the two methods is that one takes a line number and a column number for the message, and the other one takes only a line number. In the second case, the message is associated with a line, and in the first case, it is more precise and assigns the message to a particular column on the line.

The *Check* class provides a very useful mechanism to visit the AST of a node, this gives lot of facilities to the developer to develop a test to verify that the specification of a file/class is correct. However, the *Check* class does not allow to perform an algorithm and log messages/warnings after all files of the project have been visited. This is a limitation that represents an inconvenience and that will impede some tests that require warnings to be logged at the end of the visit of the project files.

The second approach is a lower level API, which is more complex to use, but that permits to log errors, warnings, and info messages on each file after all the files have been analysed. To do this, Checkstyle provides a class called *AbstractFileSetCheck*, which is situated in the following package; *com.puppycrawl.tools.checkstyle.api*. The main methods will be detailed below.

void beginProcessing(String charset) This method is called at the beginning of the analysis, in other words, the methods is called before starting analysing any file of the project. This is the right place to execute any algorithm that would require to be executed before starting to inspect the source code of the project. For instance, that's the place where variables and symbol tables initialisation should be done.

protected abstract void processFiltered(File file, List<String> lines) The method is called to process each file. The parameters are *file* and *lines*. *file* contains information about the file being inspected [41], such as its name, its path, the folder it belongs to if any, etc. The second parameter, *lines*, contains the different lines of the file being analysed. As you may have guessed, this API does not provide methods to navigate in a AST, but provides a lower way to navigate in a file, parsing each line of it. A default instance of this method is provided by the TreeWalker class, but the user is free to plug in another parser.

log(int lineNo, int colNo, String key, Object... args) and **log(int line, String key, Object... args)** are the same methods as the one detailed in the first approach using the *Check* class. As a reminder, they allow to log errors or warnings on the files to inform developers that a mistake was made

in the code.

And the last method, but not the least; **void finishProcessing()** is called when all the files have been processed. This permits to log errors once every file has been analysed and that the relevant information have been collected. This is useful when the verifications are made on information that can be spread in different files/classes. Indeed, it is possible to analyse every file once and collect relevant information with the **processFiltered()** method and then perform verifications on the collected data and log warning if necessary in the **finishProcessing()** method.

The **AbstractFileSetCheck** class provides useful method that brings to the missing functionalities in the **Check** class or in the **SSLR** API from SonarQube. However, in return the mechanism for analysing the files (Java classes) is less practical.

To conclude, the combination of the **Check** class and the **AbstractFileSetCheck** class seems to be a good solution to help to implements the different checks details in the previous section.

4.2.4 The implemented tests

This section explains the different tests implemented, their goal, their limits, the cases covered by the tests. The data structure used for each check is also detailed. Then, the different warning messages that can be logged to be shown to the developers will be described.

Specification fields verification

The first implemented test consists of verifying the good usage of specification fields. The check is detailed above in the Class and method specifications sub section on page 47. As a reminder, this test will check that the references to spec fields are correct and that the corresponding specification fields or derived fields are defined in the specification of the class or in one of its parents.

This test is realised in two different phases. The first one will collect data on the classes being analysed and store them using the ADT explained below. The second phase will read all the data stored previously and analyse them. If errors are detected, the test will log warnings to inform the developers that errors appeared in the code. The two-step analysis is due to the fact that the test should be able to check for specification fields in the parents of a class. However, the analysis does not guarantee that a file will be analysed before another one, and so, the test can't be sure that the parent classes will be analysed before the subclasses. It implies that all the data are collected first, and then analysed after each file has been inspected.

To collect the relevant information, some ADT were defined. Their goal is to store the information that will allow to detect potential errors.

The first ADT is an immutable specfield that represents a specification field. It can store information such as the name of the specfield, its type (int, String, ...), the type of the field (whether it is a spec field or a derived field) and the line number at which it is declared.

```
/**
 * Specfield is an immutable class. It represents a specification
 * field or a derived field
 *
 * @specfield name : String // The name of the specfield
 * @specfield type : String // The type of the specfield
 *                          (e.g. int, String, Point, ...)
 * @specfield fieldType : enum // The type of the specfield, either
 *                             'specfield' or 'derivedfield'
 * @specfield lineNumber : int // The line number at wich the
 *                             specfield was defined
```

```
 *
 * @invariant name not null
 *             && name not empty
 *             && name is a valid Java class name
 *             && type not null
 *             && type not empty
 *             && type represents a valid type among the project being
 *                analysed
 *             && fieldType in {specfield, derivedfield}
 *             && lineNumber >= 0
 *
 * @author Adrien Houdart
 */
public final class Specfield {

    public static final String SPECFIELD = "specfield";
    public static final String DERIVEDFIELD = "derivedfield";

    private final String name;
    private final String type;
    private final int lineNumber;
    private final String fieldType;

    // Abstraction Function:
    // AF(r) = a specfield, s, with s.name = r.name,
    //             s.type = r.type,
    //             s.fieldType = r.fieldType
    //             and s.lineNumber = lineNumber
    //
    // Representation Invariant:
    //      name not null and name is a valid Java class name
    //      type not empty
    //      SPECFIELD.equals(fieldType)
    //          || DERIVEDFIELD.equals(fieldType)
    //      lineNumber > 0
    //

    ...
}
```

Figure 4.5 – *Specfield* class specification

The second ADT defined for this test represents an occurrence of a reference to a specification field. This ADT keeps information about the name of the specfield it references as well as the line number the reference was found during the analysis.

```java
/**
 * SpecfieldRef is an immutable class. It represents a reference to a
 * specification field.
 *
 * @specfield name : String // The name of the specfield it references
 * @specfield lineNumber : int // The line number at which the
 *                             reference to the specification
 *                             was found.
 * @specfield context : String // The context on witch the specfield
 *                             is used. e.g. Circle, Point, ...
 *
 * @invariant name not null && name not empty
 *           && context not null
 *           && context not empty
 *           && lineNumber >= 0
 *
 * @author Adrien Houdart
 */
public class SpecfieldRef {

    private final int lineNumber;
    private final String name;
    private final String context;

    // Abstraction Function:
    // AF(r) = a specfield reference, s, with s.name = r.name,
    //            s.lineNumber = lineNumber
    //            and s.context = context
    //
    // Representation Invariant:
    //      name not empty
    //      context not empty
    //      lineNumber > 0
    //

    ...
}
```

Figure 4.6 – *SpecfieldRef* class specification

The third ADT represents a data type, in other words, a Java class that has been parsed. It has several fields, each one contains relevant information that will be used for the verifications. These fields are: the name of the parsed file, the name of its class, but also the set of the names of the parents of the class if any, the set of its specfields and also the list of all references to specifications fields used in the class. And finally, it contains a list of the data type of

its inner classes if any. This ADT is defined by using an abstract class called $DataType$ (that will also be used by another test) and $SpecificationDataType$ that extends the $DataType$ class.

```
/**
 * DataType is a an immutable class. It represents a data type.
 *
 * @specfield fileName : String // The name of the file linked to
 *                              the data type
 * @specfield name : String // The name of the class represented by
 *                          the data type
 * @specfield packageName[0..1] : String // The name of package of the
 *                                       class if any
 * @specfield parents : Set<String> // The set of the names of the
 *                                  parents of the class
 *
 * @invariant fileName not null && fineName not empty
 *           && fineName represents on valid file for the
 *               current project
 *           && name not null && not empty
 *           && name represents a valid Java class
 *           && packageName represents a valid package name if any
 *           && for all i in 0..n where n = parents.size-1,
 *              parents[i] is not null and not empty and represents
 *             a valid Java class name
 *
 * @author Adrien Houdart
 */
public abstract class DataType {

    private final String fileName;
    private final String name;
    private final String packageName;
    private final Set<String> parents;

    // Abstraction Function:
    // AF(r) = a data type, d, with s.fileName = r.fileName,
    //            s.name = r.name,
    //            s.packageName = r.packageName
    //            and s.parents = r.parents
    //
    // Representation Invariant:
    //      fileName not empty
    //      && name not empty
    //      && for all i in 0..n where n = parents.size-1,
    //         parents[i] != null
    //

    ...
```

```
}
```

Figure 4.7 – *DataType* class specification

```
/**
 * SpecificationDataType is a an immutable class. It represents a
 * data type about specifications..
 *
 * @specfield specfields : Set<Specfield> // The set of the
 *                                       specfields of the class
 * @specfield specfieldRefs : Set<SpecfieldRef> // The set of the
 *                                         references to specfields
 * @specfield innerClasses : Set<InnerDataType> // The set of data
 *                                             type representing inner
 *                                             classes
 *
 * @invariant for each specfields in specfields, specfields[i] is not
 *              null and not empty
 *           && for each specfieldRefs in specfieldRefs,
 *              specfieldRefs[i] is not null and not empty
 *           && for each innerClass in innerClasses, innerClass[i]
 *              is not null
 *
 * @author Adrien Houdart
 */
public class SpecificationDataType extends DataType {

   private final Set<Specfield> specfields;
    private final Set<SpecfieldRef> specfieldRefs;
    private final Set<InnerDataType> innerClasses;

    // Abstraction Function:
    // AF(r) = a data type, d, with s.fileName = r.fileName,
    //            s.name = r.name,
    //            s.packageName = r.packageName,
    //            s.parents = r.parents,
    //            s.specfields = r.specfields
    //            s.specfieldRefs = r.specfieldRefs
    //            and s.innerClasses = r.innerClasses
    //
    // Representation Invariant:
    //      fileName not empty
    //      && name not empty
    //      && for all i in 0..n where n = parents.size-1,
    //         parents[i] != null
    //      && for all i in 0..n where n = specfields.size-1,
    //         specfields[i] is not empty
    //      && for all i in 0..n where n = specfieldRefs.size-1,
```

```
    //        specfieldRefs[i] is not empty
    //     && for all i in 0..n where n = innerClasses.size-1,
           innerClasses[i] is not null
    //

    ...
}
```

Figure 4.8 – *SpecificationDataType* class specification

And finally, the last ADT defined for this test represents a data type for an inner class and extends the *DataType* class. In addition to the fields it inherits from the *DataType* class, this ADT defines additional fields; the start line of the inner class as well as its end line.

```
/**
 * InnerDataType is an immutable class. It represents an inner
 * specification data type, i.e. a data type declared in an
 * innerclass.
 *
 * @specfield startLine : int // The start line of the inner class
 * @specfield endLine : int // The end line of the inner class
 *
 * @invariant startLine > 0
 *           && endLine > 0 && endLine >= startLine
 *
 * @author Adrien Houdart
 */
public final class InnerDataType extends SpecificationDataType {

    private final int startLine;
    private final int endLine;

    // Abstraction Function:
    // AF(r) = a data type, d, with s.fileName = r.fileName,
    //            s.name = r.name,
    //            s.packageName = r.packageName,
    //            s.parents = r.parents,
    //            s.specfields = r.specfields
    //            s.specfieldRefs = r.specfieldRefs,
    //            s.innerClasses = r.innerClasses,
    //            s.startLine = r.startLine
    //            and s.endLine = r.endLine
    //
    // Representation Invariant:
    //     fileName not empty
    //     && name not empty
    //     && for all i in 0..n where n = parents.size-1,
```

```
    //            parents[i] != null
    //      && for all i in 0..n where n = specfields.size-1,
    //            specfields[i] is not empty
    //      && for all i in 0..n where n = specfieldRefs.size-1,
    //            specfieldRefs[i] is not empty
    //      && for all i in 0..n where n = innerClasses.size-1,
    //            innerClasses[i] is not null
    //      && startLine > 0
    //      && endLine > 0
    //

    ...
}
```

Figure 4.9 – *InnerDataType* class specification

All the ADT defined above are used during the analysis of the different files to collect the data. The check maintains a symbol table represented by a map where the key is the name of the class being analysed and the value is a *DataType* object representing the class.

The check will analyse each file of a Java project. For each file, it will create a *DataType* object and collect information such as the name of the class, the package it belongs to and the name of the classes or interfaces it inherits. Then, it will parse the specifications of the class and extract the specification fields and the derived fields from these specifications and store them in the *DataType* object. It will also look for all the references to specfield among the class.

During this step, the check will verify that there is no duplicate in the different fields of the class. If any, it will log a message saying that the field is duplicated. This message will contain information such as the line number and the name of the field.

If there is any inner class, it will repeat the process for these inner classes. Then, the *DataType* object will be added to the symbol table. This process will be repeated for each file of the project as said before.

Once every file of the project has been analysed, the second step of the check can start.

So, every symbol of the symbol table will be analysed for verification purpose. For each symbol, the test will check that all references to specification fields are correct. This means that the field being referenced is declared in the class specifications. If it is not the case, the test will check for each of its parents if one of them contains the field being referenced. So, from the list of parents

the data type has, it will try to get the parent from the symbol table. If the parent is in the table, it checks that the field is defined in it, and keep doing this until it found it or until there is no more parent to visit. Indeed, it is possible that parents of a class do not exist in the symbol table. It happens when the class extends a class from the Java API or when it extends a class from an external library.

The test will then look that specfields used in innerclass are defined either in the inner class (or in one of its parents) or in the outer class.

If a specfield that is referenced is not found by the check, a message is logged, telling that the field is not defined.

The figure 4.10 shows an error reported by SonarQube. Its shows that there are two fields having the same name, and warns the developers of this issue.

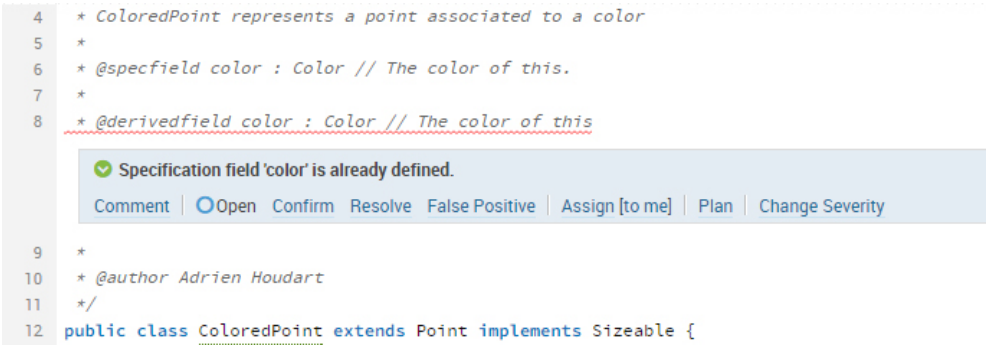

Figure 4.10 – Duplicate specification fields

The figure 4.11 shows that some specfields such as *length* and *max* are used but never defined. It is important to notice that the specfields x, y and $size$, used in the precondition of the $setMax()$ method on the second figure are not defined in the overview of the class (see 4.10) but no errors are logged for these references to specfields. Indeed, these specification fields are inherited from the *Point* class and from the *Sizeable* interface.

This test however does not behave as expected with complex generic classes such as:

```
public final class TransactionalMap<K,V extends Symbol> implements
    Transactional, Iterable<K>
```

The test is not able to figure out the parents of a class when using generics because the parser is only capable of detecting simple cases. Currently, generics are not supported, but this issue could be improved in the future.

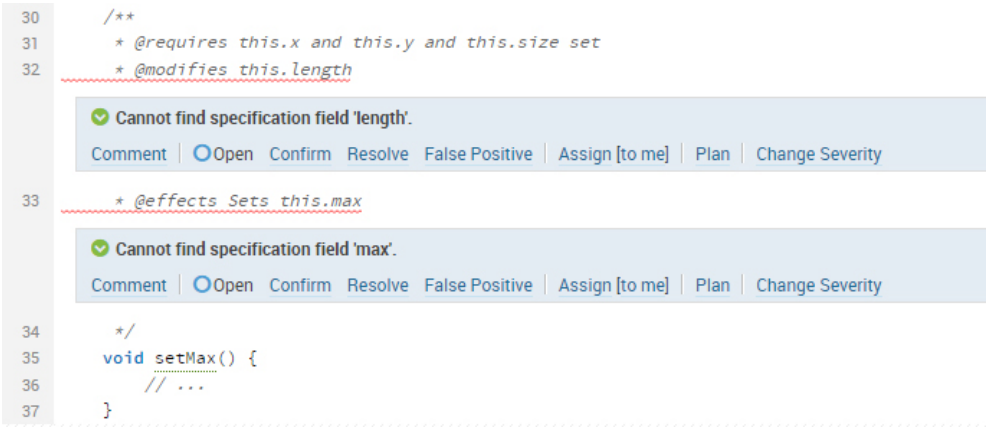

Figure 4.11 – Unknown specification fields

Modifies clause verification

The second implemented test consists of verifying the good usage of the *@modifies* clause. The check is based on some idea explained in the Class and method specifications sub section, in particular in the points talking about The mutability of a class (page 49), Constructors (page 51) and Method specifications (page 53).

The test detailed here will check classes for good usage of the $@modifies$ clauses, ensure that developers did not forget it on their method and check that immutable classes do not have method that are susceptible to mutate the object. This test also consists of two different steps where the first one collects information and the second one focus on verifying that the specifications are correct.

Bellow, the different ADT used for this test are described. They are used to collect data through the analysis of the different classes of the project, and these data will then be used to perform verifications.

The first ADT collects information about a specific method in a class. This information will then be used to determine if the specification of the method do not differ from the implementation. The information collected during the analysis are the name of the method, the content of the $@modifies$ and $@effects$ clauses, whether or not the methods override a method of its parents, the visibility of the method, whether the method is abstract or not and finally, the list of the references to statements that modifies the object the methods belongs to. This ADT is called $Method$ and is shown below.

```java
/**
 * Method is an immutable class. It represents a method and
 * information related to this method.
 *
 * @specfield name : String // The name of the method
 * @specfield modifies : String[0..1] // The content of the modifies
 *                                    clause if any
 * @specfield effects : String[0..1] // The content of the effects
 *                                    clause if any
 * @specfield override : boolean // Whether the method overrides a
 *                                method in a superclass
 * @specfield visibility : enum // The visibility of the method
 *                               (private, public, protected, package)
 * @specfield abstract : boolean // Whether the method is abstract
 *                                or not
 * @specfield modifierRefs : List<String> // The list of the
 *                                         statements that
 *                                         modifies the object
 *
 * @invariant name null and name not empty
 *           && visibility in {private, public, protected}
 *           && modifierRefs not null
 *           && visibility in {private, public, package, protected}
 *           && for all i in 0..n where n = modifierRefs.size-1,
 *              modifierRefs[i] != null
 *
 * @author Adrien Houdart
 */
public class Method {

    public static final String PRIVATE = "private";
    public static final String PUBLIC = "public";
    public static final String PACKAGE = "package";
    public static final String PROTECTED = "protected";

    private final int lineNumber;
    private final String name;
    private final String modifies;
    private final String effects;
    private final boolean override;
    private final String visibility;
    private final boolean isAbstract;
    private final List<String> modifierRefs;

    // Abstraction Function:
    // AF(r) = a method, m, with m.name = r.name,
    //            m.modifies = r.modifies,
    //            m.effects = r.effects,
    //            m.override = r.override,
```

```
    //            m.visibility = r.visibility,
    //            m.abstract = r.isAbstract,
    //            and m.modifierRefs = r.modifierRefs
    //
    // Representation Invariant:
    //      name not empty
    //      modifierRefs != null
    //      PRIVATE.equals(visibility)
    //          || PUBLIC.equals(visibility)
    //          || PACKAGE.equals(visibility)
    //          || PROTECTED.equals(visibility)
    //      for all i in 0..n where n = modifierRefs.size-1,
    //          modifierRefs[i] != null
    //

    ...
}
```

Figure 4.12 – *Method* class specification

The second ADT represents a data type that contains information about the mutability of the class, and about its methods. This ADT stores information such as the name of the file containing the class, the name of the class, its package, the list of its parents and the mutability of the class. It has also a list of *Methods*, the ADT previously detailed. This ADT is called $ModifierType$ and extends the *DataType* class that was previously detailed.

```
/**
 * ModifierType is a an immutable class. It represents a data type
 * about containing information about the mutability of a class.
 *
 * @specfield immutable : boolean // Whether or not the class is
 *                                immutable
 * @specfield methods : Set<Method> // All the methods of the data
 *                                  type
 * @specfield declarationLineNumber : int // The line at which the
 *                                  data type is declared
 *
 * @invariant methods != null
 *           && for all i in 0..n where n = methods.size-1,
 *               methods[i] != null
 *           && declarationLineNumber >= 0
 *
 * @author Adrien Houdart
 */
public class ModifierType extends DataType {

    private final Boolean immutable;
    private final Set<Method> methods;
    private final int declarationLineNumber;

    // Abstraction Function:
    // AF(r) = a data type, d, with s.fileName = r.fileName,
    //            s.name = r.name,
    //            s.packageName = r.packageName,
    //            s.parents = r.parents,
    //            s.immutable = r.immutable
    //            s.methods = r.methods
    //            and s.declarationLineNumber = r.declarationLineNumber
    //
    // Representation Invariant:
    //      fileName not empty
    //      && name not empty
    //      && for all i in 0..n where n = parents.size-1,
    //            parents[i] != null
    //      && methods != null
    //      && for all i in 0..n where n = methods.size-1,
    //           methods[i] != null
    //      && declarationLineNumber >= 0
    //

    ...
}
```

The check maintains a symbol table with the information collected during the

Figure 4.13 – *ModifierType* class specification

analysis. This table is represented by a map where the key is the name of the class and the value is a *ModifierType* object.

During the first phase, the check collects information such as the name of the class, the package it belongs to and its parents. It also checks if the class is mutable or not, and stores this piece of information. This is done by parsing the description of the ADT, and it looks for the *immutable* or *mutable* keyword. If that information is not present, the class is considered as mutable. Then, it looks through all methods and gathers information that will be relevant for later verifications.

The second phase of the test consists of verifications. The test verifies that each method that has a @*modifies* clause has also an @*effects* clause and conversely. If the @*effects* is forgotten, the check will warn the developer as shown in the figure 4.14.

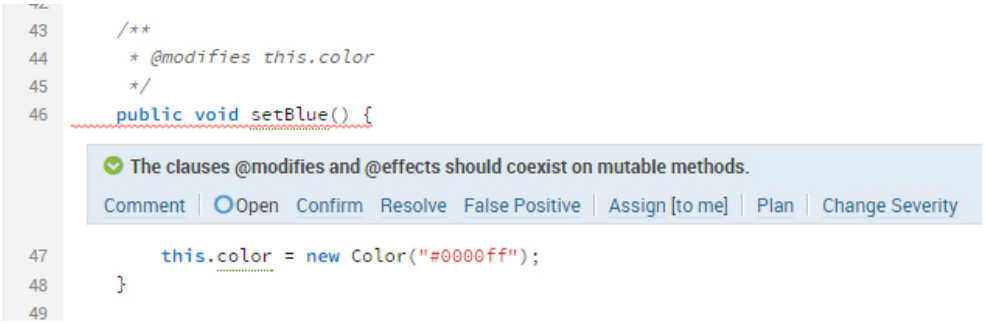

Figure 4.14 – Missing @*effects* clause

Then it checks if the class is immutable, in that case, it verifies that there is no @*modifies* clause on each method of the class. Indeed, methods of an immutable class should not modify the state of the ADT represented by this class. The figure 4.15 shows an issue reported by the test that violates this rule.

After that, the next verification inspects that the mutability of its parents is the same as the mutability defined in the class. The mutability must always be the same between the sub and super classes. If the mutability changed through the inheritance of classes, a message will inform that a mistake was made. This case is illustrated on the figure 4.16.

CheckStyle logs error messages for each error found, each one indicates the name of the method and the line number affected by this error.

Finally the test checks that the implementation of the methods of an im-

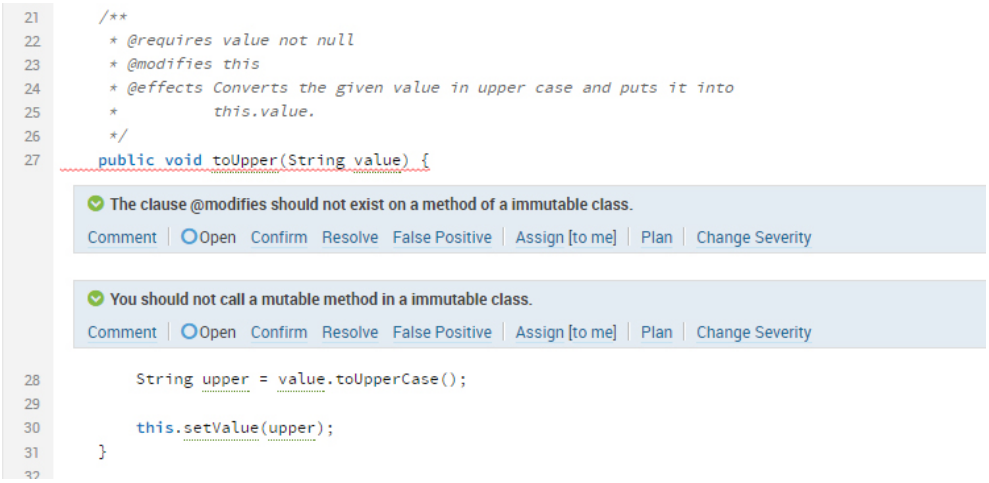

Figure 4.15 – Mutator method on immutable class

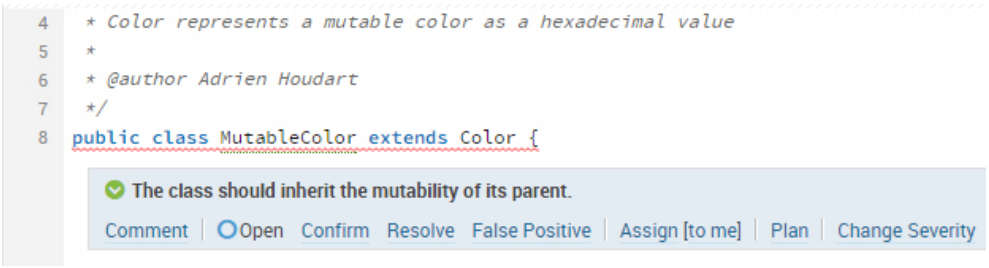

Figure 4.16 – Mutability changed through inheritance

mutable class, or the implementation of a class that has no @*modifies* clause does not modify the object. To do this, the check looks for calls to method that have a @*modifies* clause. If the test finds such methods, it warns the developer that the implementation does not respect the specification of the class/method. It also checks for calls to methods that starts with “set”, “add” and “remove”, that can be considered as mutator methods. Then, it checks for mutation of instance variables such as on the figure 4.17 where the @*modifies* clause is missing despite that the instance variable *color* is being modified. When dealing with an immutable class, the test detects when instance variables are modified and notifies the developers of this non-conform behaviour as shown on the figure 4.18.

If the method overrides a method from its parent, the check looks for the specification of its ancestor and check if the method specification has a @*modifies* clause. If it is not the case, a message is logged to warn the developer that the @*modifies* clause should exist on the super class.

The figure 4.19 illustrates a method defined in the super class, where no @*modifies* clause is present. And the figure 4.20 illustrate a sub class, where the method is overridden, and where the implementation mutates the object.

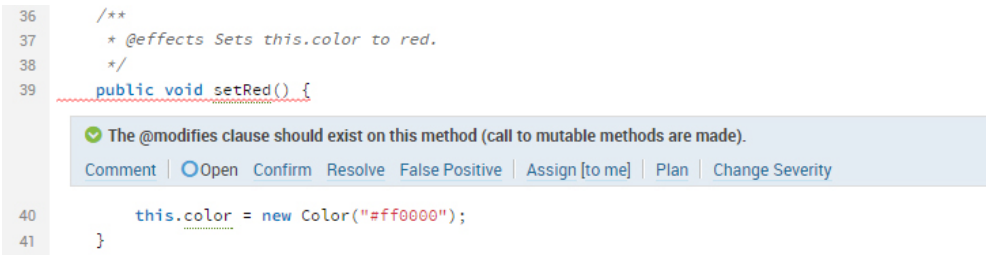

Figure 4.17 – Missing @*modifies* clause

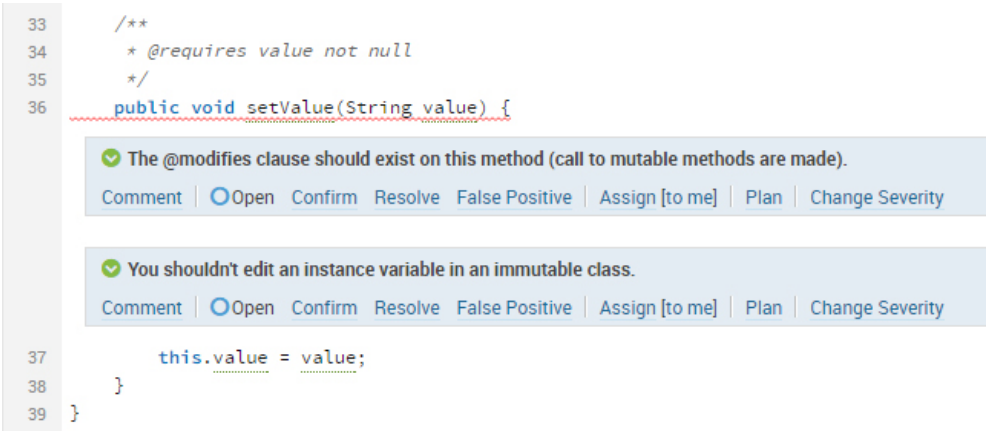

Figure 4.18 – Instance variable modified in immutable class

Because of this, the check logs a warning to tell that the specification of the method should specify that the method modifies something.

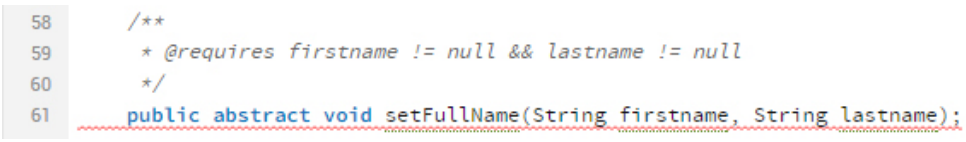

Figure 4.19 – Method defined on the super class

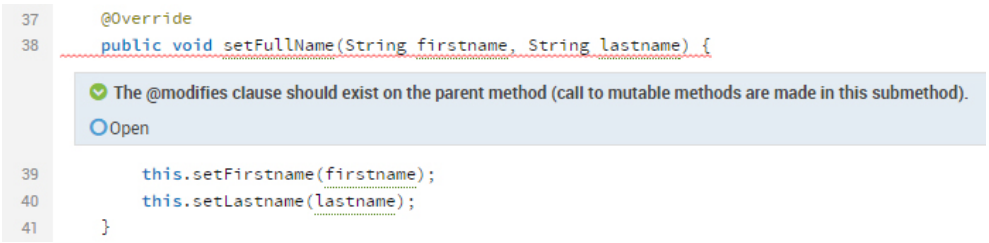

Figure 4.20 – Method implemented in the sub class

The test explained in this sub section can detect a few cases where the @*modifies* annotation may be misused. It will help developers to improve the quality of their code. However, the test does not cover all the cases detailed in the Class and method specifications sub section, and thus could be improved in the fu-

ture. Moreover, the test can raise false positives, which is not a problem if the number of false positives is not too important in comparison with the number of correct detections.

4.2.5 Summary

The two different checks explained in this section will help developers to increase the quality of their programs. Moreover, it will help code reviewers when analysing the work done by their colleagues because it will detect common errors made by developers, and do this automatically. Currently, this task is being performed manually, an thus, reviewers can miss some errors. A simple report will be generated by SonarQube and Checkstyle, as shown on the figures used in the last subsection. However, several tests could be added to go further in error detections. These tests are detailed in the first section of this chapter. False positives will be detected by the different tests, as said before this is not a problem, because the messages logged by the checks are for information purpose, and developers have to correct the errors found by the tool manually. Nonetheless, if false positives are detected too frequently, the test should be modified to improve the wrong detections. Nevertheless, the different tests implemented for this thesis constitute a first prototype. This prototype can be improved and used daily in the development team of Skalup, or by any developer using specifications in the same way as Skalup does.

The following table summarizes the different tests implemented in the prototype.

Table 4.2 – Summary of the different implemented tests

Category	Test	Summary	Impl.
Usage of specification fields and derived fields	1	Existence of specfields and derived-fields	V
	2	Access of specfields on inner classes	V
	3	Chaining specfields	V
The mutability of a class	4	Mutability of methods in immutable classes	V
	5	Call to mutator methods in immutable class	V
	6	equals & hashcode	X
	7	Representation exposure	X
Constructor	8	Postcondtions of constructors (@effects & @modifies)	X
	9	Defensive checks of preconditions	X
	10	Suggestion of templates for specifications on constructors of ADT	X
Method specifications	11	Defensive checks of preconditions (same as the test 9)	X
	12	Omission of specification tags (e.g. @modifies, @effects, @throws, etc.)	V
	13	Params appears in the post condition (@modifies, @effects, @return and @throw tags)	X
	14	Chaining params (similar as the test 3)	X
	15	Old & Result keywords	X
Abstraction Functions	16	Presence of the specfields and the representation fields in the AF	X
	17	Type of comments for the AF definition	X
Representation Invariants	18	Coexistence of the Representation Invariant and the checkRep() method	X
	19	Call to the checkRep() methods when needed	X
	20	Extraction of information from the Representation Invariant	X

CHAPTER 5

CONCLUSION

In this chapter, I present a brief summary of the work I accomplished for this master thesis. Then I present a critical outlook and finally I discuss about future work and researches that could be conducted based on my findings.

5.1 General conclusion

There exists some tools that permit to integrate the concept of Design by Contract into different programming languages. However, the way specifications are used among the development team of Skalup, does not permit to integrate one of these tools without having a deep impact on how the developers work. This is the reason that conducted me to implement a new prototype specific to the way specifications are use among Skalup.

Based on existing tools and my knowledges of the specification used by Skalup, I determined checks that can be executed on the source code of an application. These checks aim to reduce the number of errors introduced by developers in their specifications or source code.

The main difficulty in performing code analysis tool is the lack of documentation on tools providing mechanisms to achieve such tools. Using CheckStyle together with SonarQube for such prototype is therefore a maturely considered decision, since these tools are widely used in the software industry, there are therefore benefiting from better documentation and a larger community.

I highlighted several verifications that can be performed to improve the correctness between specifications and their implementations. However, only a few of them have been integrated in the prototype I realized.

Moreover the implemented tests have limits, as explained in this thesis. These limitations can be improved to increase the rate of detection of errors in the specifications.

To conclude, this master thesis shows an effective methodology to perform verifications on specifications. A prototype permits to integrate some of the tests detailed in this thesis, and can be easily extended with other verifications.

The development team of Skalup is satisfied by the prototype which implies that it is worth continuing the realisation of such a tool.

5.2 Future Work

This section outlines directions for future researches, discusses the findings and contributions and points out the limitations of the current prototype.

First, the tests detailed in the chapter 4 are not all implemented. Only a few of them are supported by the prototype. In the future, more checks can be implemented and integrated to the tool. The list of the unimplemented test is detailed in the table 4.2. Integrating the prototype into IDEs could also be an interesting topic as it would help developers while they are coding.

Moreover, the implemented tests may detect false positives. It could be interesting to determine the ratio between these false positives and correct detections. If there are too many false positives, the tests should be improved to make them more precise. An evaluation can be conducted to determine the usefulness of the tool and its results.

Another improvement could be to improve the way files are parsed with the API used. Indeed, the way Checkstyle was used for this prototype is not optimal for analysing easily files with two different phases as explained in this thesis. A way to enhance this API would be to develop a new layer over this API to facilitate the interaction with Checkstyle. This layer can introduce an AST (abstract syntax tree) visit based on a grammar, where Java classes will be parsed and an AST will be created, containing different nodes. These nodes can then be visited to extract relevant information. This approach is easier to use and easier to maintain than the approach used in this thesis where each line of the file has to be parsed manually. There already exists tools that provide such functionalities. For instance, Antlr, which is widely used in the Java world and has a community, may be a good starting point. Furthermore, Antlr provides existing grammar among which Java 7 and Java 8. These grammars have already proven themselves and can then be trusted and used for such a tool in the future.

Furthermore, deeper researches could determinate new verifications that could be executed on the source code to enlarge the field of errors that can be detected. Or introduce new way to support Design by Contract into Java, such as the generation of assertions to verify precondition based on the specifications.

The most important step that could be, and will be done, is the integration of the prototype with the codebase of Skalup, and integrating it in the development lifecycle. Skalup developers use continuous integration, so this tool will be used daily by developers to verify their code. The daily use of this tool will permit to receive lots of feedbacks. These feedbacks could be analysed in the future to increase the rate of detection of the prototype.

It would be interesting to set up an evaluation focused on users, that would determine how the tool is being used, if it meets their expectations and establish new ways, according to them, that would allow to improve the usability of the tool.

All the points listed above will make the tool more effective.

BIBLIOGRAPHY

[1] "Modern jass." http://modernjass.sourceforge.net/. Accessed: 2016-03-18.

[2] "Software design and implementation." https://courses.cs.washington.edu/courses/cse331/11wi/conceptual-info/specifications.html. Accessed: 2016-06-08.

[3] "About eiffel software." https://www.eiffel.com/company/about-us/. Accessed: 2016-01-21.

[4] B. Meyer, *Design by contract*. Prentice Hall, 2002.

[5] B. Meyer, *Design by Contract*. Software Engineering Inc., 1986.

[6] B. Liskov, *Keynote address-data abstraction and hierrarchy*. 1988.

[7] B. Meyer, J.-M. Nerson, and M. Matsuo, *Eiffel: Object-oriented design for software engineering*. Software Engineering Inc., 1986.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[9] "The daikon dynamic invariant detector." https://plse.cs.washington.edu/daikon/. Accessed: 2016-03-14.

[10] G. T. Leavens and Y. Cheon, "Design by contract with jml," 2006.

[11] "The java modeling language." http://www.eecs.ucf.edu/~leavens/JML//index.shtml. Accessed: 2016-07-05.

[12] "Openjml." http://www.openjml.org/. Accessed: 2016-07-05.

[13] J. Rieken, "Design by contract for java-revised," *Master's thesis, Department für Informatik, Universität Oldenburg*, 2007.

[14] "Spingcontracts." http://springcontracts.sourceforge.net/pg000.html. Accessed: 2016-07-05.

[15] M. Fähndrich, "Static verification for code contracts," in *International Static Analysis Symposium*, pp. 2–5, Springer, 2010.

[16] N. M. Lê, "Contracts for java: A practical framework for contract programming," tech. rep., Technical report, Google Switzerland GmbH, 2011.

[17] "Cofoja." https://github.com/nhatminhle/cofoja. Accessed: 2016-07-06.

[18] "Bean validation." http://beanvalidation.org. Accessed: 2016-07-06.

[19] l. GuÃľrub, "Java bean validation api," January 2014.

[20] "Valid4j." http://www.valid4j.org. Accessed: 2016-07-06.

[21] "Contract4j." https://deanwampler.github.io/contract4j/. Accessed: 2016-07-06.

[22] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[23] A. Asperti, "The intensional content of rice's theorem," in *ACM SIGPLAN Notices*, vol. 43, pp. 113–119, ACM, 2008.

[24] "Checkstyle." http://checkstyle.sourceforge.net. Accessed: 2016-07-07.

[25] "Github checkstyle." https://github.com/checkstyle/checkstyle. Accessed: 2016-07-07.

[26] "Siia codie awards 2007 winners." https://www.siia.net/codie/About-the-Awards/Past-Winners/2007-Winners. Accessed: 2016-06-08.

[27] "Parasoft jtest wins second consecutive infoworld technology of the year award." https://www.parasoft.com/press/parasoft-jtest-wins-second-consecutive-infoworld-technology-of-the-year-award/. Accessed: 2016-02-02.

[28] "Coverity ceo named by mit technology review." http://www.coverity.com/press-releases/press_story67_08_19_08/. Accessed: 2016-07-10.

[29] "Coverity named one of the fastest growing companies." http://www.prnewswire.com/news-releases/coverity-named-one-of-the-fastest-growing-companies-in-north-america-on-deloittes-2011-technology-fast-500-132245658.html. Accessed: 2016-07-10.

[30] "Coverity named one of the fastest growing companies 2009." http://www.coverity.com/press-releases/coverity_named_deloitte_fast_500/. Accessed: 2016-07-10.

[31] "Coverity wins siia codie." http://www.coverity.com/press-releases/coverity-wins-siia-codie-award-for-best-software-development-solution/. Accessed: 2016-07-10.

[32] "Coverity®- static code analysis." http://www.synopsys.com/software/coverity/Pages/default.aspx. Accessed: 2016-07-10.

[33] G. Vishal, L. Sean, S. Xiang, W. Guo-Shiuan, W. Bradley, and Z. Pengfei, "Analysis tool evaluation: Grammatech codesonar." https://www.cs.cmu.edu/~aldrich/courses/654-sp07/tools/garg-codesonar-07.pdf, April 2007.

[34] B. Liskov and J. Guttag, *Program development in JAVA: abstraction, specification, and object-oriented design*. Pearson Education, 2000.

[35] "Writing abstraction functions and rep invariants." https://courses.cs.washington.edu/courses/cse331/11wi/conceptual-info/abstraction-functions-and-rep-invariants.html. Accessed: 2016-06-08.

[36] P. Heymans, "Conception et programmation orientÃľes-objet," September 2014.

[37] "Collections." http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Collections.html#unmodifiableList(java.util.List). Accessed: 2016-07-10.

[38] "Object - hashcode() (java platform se 7)." http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode(). Accessed: 2016-07-02.

[39] "Object - equals(...) (java platform se 7)." http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object). Accessed: 2016-07-02.

[40] B. Liskov and J. Guttag, *Program development in JAVA: abstraction, specification, and object-oriented design*, ch. 7, p. 182. Pearson Education, 2000.

[41] "File (java platform se 7)." https://docs.oracle.com/javase/7/docs/api/java/io/File.html. Accessed: 2016-06-23.

APPENDIX A

APPENDICES

This chapter presents the main java classes implemented to realize the different checks. They use the CheckStyle API to make a SonarQube plugin. They are part of a prototype, meaning that the different classes are not optimized. They may also contain bugs or may not be entirely specified. The source of the project can be freely downloaded from the repository located at `https://bitbucket.org/adrienhoudart/liskov-checkstyle`

A.1 DataType.java

```
package be.adrienhoudart.tool.domain;

import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 * DataType represents a mutable data type.
 *
 * @specfield fileName : String // The name of the file linked to the
 *                              data type
 * @specfield name : String // The name of the class represented by the
 *                          data type
 * @specfield packageName : String // The name of package of the class
 * @specfield parents : Set<String> // The set of the names of the parents
 *                                  of the class
 * @specfield specfields : Set<Specfield> // The set of the specfields of the
 *                                    class
 * @specfield specfieldRefs : List<SpecfieldRef> // The list of the references
 *                                           to specfields
 * @specfield innerClass : List<InnerDataType> // The list of data
 *                                          type representing inner
 *                                          classes
 *
 * @invariant fileName not empty
 *          && name not empty
 *
 * @author aho
 */
public class DataType {

    private static final Set<String> ALLOWED;

    static {
        ALLOWED = new LinkedHashSet<>();
        ALLOWED.add("class");
    }

    private final String fileName;
    private String name;
```

```
private String packageName;
private final Set<String> parents = new LinkedHashSet<>();
private final Set<Specfield> specfields = new LinkedHashSet<>();
private final List<SpecfieldRef> specfieldRefs = new LinkedList<>();
private final List<InnerDataType> innerClass = new LinkedList<>();

public DataType(String fileName) {
    this.fileName = fileName;
}

public String getFileName() {
    return fileName;
}

public String getName() {
    return name;
}

public Set<String> getParents() {
    return parents;
}

public Set<Specfield> getSpecfields() {
    return specfields;
}

public String getPackageName() {
    return packageName;
}

public List<InnerDataType> getInnerClass() {
    return innerClass;
}

public void addSpecfield(Specfield specfield) {
    specfields.add(specfield);
}

public boolean addParent(String parent) {
    return parents.add(parent);
}

public List<SpecfieldRef> getSpecfieldRefs() {
    return specfieldRefs;
```

```
}

public void addSpecfieldRef(int lineNo, String ref) {
    specfieldRefs.add(new SpecfieldRef(lineNo, ref));
}

public void addInnerClass(InnerDataType dt) {
    innerClass.add(dt);
}

public void setName(String name) {
    this.name = name;
}

public void setPackageName(String packageName) {
    this.packageName = packageName;
}

public boolean belongsToInner(int lineNo) {
    for (InnerDataType innerClass : innerClass) {
        if (lineNo >= innerClass.getStartLine() && lineNo <=
            innerClass.getEndLine()) {
            return true;
        }
    }

    return false;
}

@Override
public String toString() {
    return "DataType{" + "name=" + name + ", parents=" + parents
           + ", specfields=" + specfields + ", specfieldRefs="
           + specfieldRefs + '}';
}

////////////////////
// HELPER METHODS
////////////////////

public boolean isDefined(SpecfieldRef ref, Map<String, DataType> symTable) {
    return isDefined(ref.getName(), symTable);
}
```

```
    public boolean isDefined(String name, Map<String, DataType> symTable) {

        if (ALLOWED.contains(name)) {
            return true;
        }

        for (Specfield specfield : specfields) {
            if (specfield.getName().equals(name)) {
                return true;
            }
        }

        for (String parent : parents) {
            if (symTable.containsKey(parent)
                    && symTable.get(parent).isDefined(name, symTable)) {
                return true;
            }
        }

        return false;
    }

    public boolean isDefined(Specfield sf, Map<String, DataType> symTable) {
        return isDefined(sf.getName(), symTable);
    }
}
```

Figure A.1 – *DataType* class

A.2 InnerDataType.java

```java
package be.adrienhoudart.tool.domain;

/**
 * InnerDataType represents a mutable inner data type.
 *
 * @specfield startLine : int // The start line of the inner class
 * @specfield endLine : int // The end line of the inner class
 *
 * @invariant startLine > 0
 *            && endLine > 0 && endLine >= startLine
 *
 * @author Adrien Houdart
 */
public final class InnerDataType extends DataType {

    private int startLine;
    private int endLine;

    public InnerDataType(String fileName) {
        super(fileName);
    }

    public int getStartLine() {
        return startLine;
    }

    public int getEndLine() {
        return endLine;
    }

    public void setStartLine(int startLine) {
        this.startLine = startLine;
    }

    public void setEndLine(int endLine) {
        this.endLine = endLine;
    }
}
```

Figure A.2 – *InnerDataType* class

A.3 Modifier.java

```
package be.adrienhoudart.tool.domain;

import java.util.LinkedList;
import java.util.List;

/**
 * Modifier represents a immutable data type that contains information related
 * to the mutability of a method.
 *
 * @specfield name : String // The name of the method
 * @specfield modifies : String[0..1] // The content of the modifies
 *                                    clause if any
 * @specfield effects : String[0..1] // The content of the effects
 *                                    clause if any
 * @specfield override : boolean // Whether the method overrides a method in
 *                               in a superclass
 * @specfield visibility : enum[0..1] // The visibility of the method (private,
 *                              public, protected)
 * @specfield abstract : boolean // Wheter the method is abstract or not
 * @specfield modifierRefs : List<String> // The list of the references to
 *                                        statements that modifies the
 *                                        object
 *
 * @invariant name not empty
 *          && visibility in {private, public, protected}
 *
 * @author Adrien Houdart
 */
public class Modifier {

    public static final String PRIVATE = "private";
    public static final String PUBLIC = "public";
    public static final String PACKAGE = "package";
    public static final String PROTECTED = "protected";

    private int lineNo;
    private String name;
    private String modifies;
    private String effects;
    private boolean override;
    private String visibility;
    private boolean abstractt = false;
```

```
private final List<String> modifierRefs = new LinkedList<>();

public Modifier() {
}

public int getLineNo() {
    return lineNo;
}

public String getName() {
    return name;
}

public String getModifies() {
    return modifies;
}

public String getEffects() {
    return effects;
}

public String getVisibility() {
    return visibility;
}

public boolean isAbstract() {
    return abstractt;
}

public void setLineNo(int lineNo) {
    this.lineNo = lineNo;
}

public void setName(String name) {
    this.name = name;
}

public void setModifies(String modifies) {
    this.modifies = modifies;
}

public void setEffects(String effects) {
    this.effects = effects;
}
```

```
    public void setOverride(boolean override) {
        this.override = override;
    }

    public void setVisibility(String visibility) {
        this.visibility = visibility;
    }

    public void setAbstract(boolean abstractt) {
        this.abstractt = abstractt;
    }

    public void addModifiers(String modifier) {
        this.modifierRefs.add(modifier);
    }

    public boolean hasModifies() {
        return this.modifies != null;
    }

    public boolean hasEffects() {
        return this.effects != null;
    }

    public boolean isOverride() {
        return override;
    }

    public List<String> getModifiers() {
        return modifierRefs;
    }

    @Override
    public String toString() {
        return "ModifierRef{" + "lineNo=" + lineNo + ", name=" + name
                + ", modifies=" + modifies + ", effects=" + effects
                + ", override=" + override + ", modifiers=" + modifierRefs + '}';
    }
}
```

Figure A.3 – *Modifier* class

A.4 ModifierType.java

```
package be.adrienhoudart.tool.domain;

import be.adrienhoudart.common.util.Assert;
import java.util.List;
import java.util.Set;

/**
 * ModifierType represents a mutable data type containing information about
 * the mutability of a class.
 *
 * @specfield fileName : String // The name of the file linked to the
 *                              data type
 * @specfield name : String // The name of the class represented by the
 *                          data type
 * @specfield packageName : String // The name of package of the class
 * @specfield parents : Set<String> // The set of the names of the parents
 *                                  of the class
 * @specfield immutable : boolean // Whether or not the class is immutable
 * @specfield specfieldRefs : List<ModifierdRef> // The list of the references
 *                                               to modifiers
 *
 * @invariant fileName not empty && name not empty
 *
 * @author Adrien Houdart
 */
public class ModifierType {

    private final String fileName;
    private String name;
    private String packageName;
    private final Set<String> parents = new LinkedHashSet<>();
    private Boolean immutable = null;
    private final List<Modifier> methodRef = new LinkedList<>();
    private int declarationLineNo;

    public ModifierType(String fileName) {
        this.fileName = fileName;
    }

    public String getFileName() {
        return fileName;
    }
```

```
public String getName() {
    return name;
}

public String getPackageName() {
    return packageName;
}

public Set<String> getParents() {
    return parents;
}

public int getDeclarationLineNo() {
    return declarationLineNo;
}

public Boolean isImmutable() {
    return immutable;
}

public boolean hasMethod(String methodName) {

    for (Modifier ref : this.methodRef) {
        if (ref.getName().equals(methodName)) {
            return true;
        }
    }
    return false;
}

public Modifier getMethod(String methodName) {

    for (Modifier ref : this.methodRef) {
        if (ref.getName().equals(methodName)) {
            return ref;
        }
    }
    return null;
}


public Boolean hasImmutableValue() {
    return immutable != null;
```

```
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setPackageName(String packageName) {
        this.packageName = packageName;
    }

    public void setImmutable(Boolean mutable) {
        this.immutable = mutable;
    }

    public void setDeclarationLineNo(int declarationLineNo) {
        this.declarationLineNo = declarationLineNo;
    }

    public void addParent(String parent) {
        Assert.notNull(parent);

        this.parents.add(parent);
    }

    public void addParents(Set<String> parents) {
        Assert.notNull(parents);

        this.parents.addAll(parents);
    }

    public void addModifyingMethods(Modifier method) {
        Assert.notNull(method);

        this.methodRef.add(method);
    }

    public List<Modifier> getMethods() {
        return this.methodRef;
    }
}
```

Figure A.4 – *ModifierType* class

A.5 Specfield.java

```
package be.adrienhoudart.tool.domain;

import java.util.Objects;

/**
 * Specfield represents a mutable specification field or a derived field
 *
 * @specfield name : String // The name of the specfield
 * @specfield type : String // The type of the specfield (e.g. int, string, ...)
 * @specfield fieldType : enum // The type of the specfield, either 'specfield'
 *                             or 'derived field'
 * @specfield lineNumber : int // The line number at wich the specfield was
 *                             defined
 *
 * @invariant name not empty
 *           && type not empty
 *           && fieldType in {specfield, derivedfield}
 *           && lineNumber >= 0
 *
 * @author Adrien Houdart
 */
public final class Specfield {

    public static final String SPECFIELD = "specfield";
    public static final String DERIVEDFIELD = "derivedfield";

    private String name;
    private String type;
    private int lineNumber;
    private String fieldType;


    public Specfield() {
    }

    public String getName() {
        return name;
    }

    public String getType() {
        return type;
    }
```

```
public int getLineNumber() {
    return lineNumber;
}

public String getFieldType() {
    return fieldType;
}

public Specfield setName(String name) {
    this.name = name;
    return this;
}

public Specfield setType(String type) {
    this.type = type;
    return this;
}

public Specfield setLineNumber(int lineNumber) {
    this.lineNumber = lineNumber;
    return this;
}

public Specfield setFieldType(String fieldType) {
    this.fieldType = fieldType;
    return this;
}

@Override
public int hashCode() {
    int hash = 5;
    hash = 73 * hash + Objects.hashCode(this.name);
    hash = 73 * hash + Objects.hashCode(this.type);
    hash = 73 * hash + this.lineNumber;
    hash = 73 * hash + Objects.hashCode(this.fieldType);
    return hash;
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
```

```
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Specfield other = (Specfield) obj;
        if (!Objects.equals(this.name, other.name)) {
            return false;
        }
        if (!Objects.equals(this.type, other.type)) {
            return false;
        }
        if (this.lineNumber != other.lineNumber) {
            return false;
        }
        if (!Objects.equals(this.fieldType, other.fieldType)) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

Figure A.5 – *Specfield* class

A.6 SpecfieldRef.java

```java
package be.adrienhoudart.tool.domain;

/**
 * SpecfieldRef represents a immutable reference to a specification field.
 *
 * @specfield name : String // The name of the specfield it references
 * @specfield lineNumber : int // The line number at which the reference
 *                             to the specification was found.
 *
 * @invariant name not empty
 *           && lineNumber >= 0
 *
 * @author Adrien Houdart
 */
public class SpecfieldRef {

    private final int lineNo;
    private final String name;

    public SpecfieldRef(int lineNo, String name) {
        this.lineNo = lineNo;
        this.name = name;
    }

    public int getLineNo() {
        return lineNo;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "SpecfieldRef{" + "lineNo=" + lineNo + ", name=" + name + '}';
    }
}
```

Figure A.6 – *SpecfieldRef* class

A.7 SpecfieldCheck.java

```
package be.adrienhoudart.tools.checkspec.checks;

import be.adrienhoudart.tool.domain.DataType;
import be.adrienhoudart.tool.domain.InnerDataType;
import be.adrienhoudart.tool.domain.Specfield;
import be.adrienhoudart.tool.domain.SpecfieldRef;
import be.adrienhoudart.tools.checkspec.util.ParseUtils;
import static
    be.adrienhoudart.tools.checkspec.util.ParseUtils.extractPackageName;
import static
    be.adrienhoudart.tools.checkspec.util.ParseUtils.findLastLineOfBlock;
import static be.adrienhoudart.tools.checkspec.util.ParseUtils.goDown;
import static be.adrienhoudart.tools.checkspec.util.ParseUtils.goUp;
import static be.adrienhoudart.tools.checkspec.util.ParseUtils.isComment;
import static be.adrienhoudart.tools.checkspec.util.Regexes.COMMENT_BEGIN;
import static be.adrienhoudart.tools.checkspec.util.Regexes.COMMENT_CONTENT;
import static be.adrienhoudart.tools.checkspec.util.Regexes.COMMENT_END;
import static be.adrienhoudart.tools.checkspec.util.Regexes.COMMENT_LINE;
import static be.adrienhoudart.tools.checkspec.util.Regexes.DATA_TYPE;
import static be.adrienhoudart.tools.checkspec.util.Regexes.DERIVEDFIELD;
import static be.adrienhoudart.tools.checkspec.util.Regexes.DERIVEDFIELD_TYPE;
import static be.adrienhoudart.tools.checkspec.util.Regexes.EXTENDS;
import static be.adrienhoudart.tools.checkspec.util.Regexes.IMPLEMENTS;
import static be.adrienhoudart.tools.checkspec.util.Regexes.SPECFIELD;
import static be.adrienhoudart.tools.checkspec.util.Regexes.SPECFIELD_REF;
import static be.adrienhoudart.tools.checkspec.util.Regexes.SPECFIELD_TYPE;
import com.puppycrawl.tools.checkstyle.api.AbstractFileSetCheck;
import java.io.File;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * Checks specification and derived specification fields are defined in the
 * overview of the data type or in the overview of the parent data type. Also,
 * they should be defined only once.
 *
 * Notes:
```

```
 *
 * - Extended class and first implemented interface are also considered for the
 * definition of spec fields, however if there is more than one implemented
 * interface, the subsequent ones are not considered.
 *
 * - Extended class and first implemented interface are considered as long as
 * their definition, e.g., extends Foo, are on the same line. If there is a line
 * break between the keyword, i.e., extends or implements, and the class or
 * interface name, then it is not considered.
 *
 * - Inner classes are not handled. Moreover spec fields defined on inner
 * classes are also considered as spec fields of the outer class and vice versa.
 * This is not correct and should be fixed in a future iteration.
 *
 * - As a convenience the spec field 'class' is always considered as defined.
 *
 * @author aho
 */
public class SpecfieldCheck extends AbstractFileSetCheck {

    private static final Log LOG = LogFactory.getLog(
            SpecfieldCheck.class);

    private final Map<String, DataType> symTable;

    public SpecfieldCheck() {
        super();

        symTable = new LinkedHashMap<>();
        setFileExtensions(new String[]{"java"});
    }

    @Override
    public void beginProcessing(String charset) {
        symTable.clear();
    }

    @Override
    public void finishProcessing() {
        for (DataType dt : symTable.values()) {
            getMessageCollector().reset();
            getMessageDispatcher().fireFileStarted(dt.getFileName());

            List<SpecfieldRef> refs = dt.getSpecfieldRefs();
```

```
        for (SpecfieldRef ref : refs) {

            if (!dt.isDefined(ref, symTable)) {
                log(ref.getLineNo(), "specfield.undefined", ref.getName());
            }
        }

        for (InnerDataType inner : dt.getInnerClass()) {

            List<SpecfieldRef> innerRefs = inner.getSpecfieldRefs();
            for (SpecfieldRef ref : innerRefs) {

                if (!inner.isDefined(ref, symTable)) {
                    log(ref.getLineNo(), "specfield.undefined", ref.getName());
                }
            }

        }

        if (!refs.isEmpty()) {
            fireErrors(dt.getFileName());
        }
        getMessageDispatcher().fireFileFinished(dt.getFileName());
    }
}

@Override
protected void processFiltered(File file, List<String> lines) {
    DataType dt = new DataType(file.getAbsolutePath());

    if (ParseUtils.isEnum(lines)) {
        return;
    }

    String packageName = extractPackageName(lines);
    if (packageName != null) {
        dt.setPackageName(packageName);
    }

    extractClassInfo(dt, lines);

    if (dt.getName() == null) {
        return;
    }
```

```
    extractSpecfields(dt, lines);
    extractInnerClass(dt, lines);

    extractSpecfieldRefs(dt, lines);

    symTable.put(dt.getName(), dt);
}

////////////////////
// HELPER METHODS
////////////////////
private void extractSpecfields(DataType dt, List<String> lines) {
    int lineNo = 1;
    boolean comment = false;

    for (String l : lines) {
        if (!comment && COMMENT_BEGIN.matcher(l).find()) {
            comment = true;
        }

        if (comment && COMMENT_CONTENT.matcher(l).find()) {
            extractSpecfield(dt, lineNo, l);
        }

        if (comment && COMMENT_END.matcher(l).find()) {
            comment = false;

            if (!dt.getSpecfields().isEmpty()) {
                return;
            }
        }

        lineNo++;
    }
}

private void extractSpecfieldRefs(DataType dt, List<String> lines) {
    int lineNo = 1;
    boolean comment = false;

    for (String l : lines) {

        if (!dt.belongsToInner(lineNo)) {
```

```
            if (!comment && COMMENT_BEGIN.matcher(l).find()) {
                comment = true;
            }

            if (comment || COMMENT_LINE.matcher(l).find()) {
                extractSpecfieldRef(dt, lineNo, l);
            }

            if (comment && COMMENT_END.matcher(l).find()) {
                comment = false;
            }
        }

        lineNo++;
    }
}

private void extractSpecfield(DataType dt, int lineNo, String line) {
    Matcher m = SPECFIELD.matcher(line);
    if (m.find()) {
        Specfield sf = new Specfield();
        sf.setName(m.group(3));
        sf.setLineNumber(lineNo);

        Matcher mType = SPECFIELD_TYPE.matcher(line);
        if (mType.find()) {
            sf.setFieldType(mType.group(7));
        }

        sf.setType(Specfield.SPECFIELD);

        if (dt.isDefined(sf, symTable)) {
            log(lineNo, "specfield.duplicate", sf.getName());
        } else {
            dt.addSpecfield(sf);
        }
    }

    m = DERIVEDFIELD.matcher(line);
    if (m.find()) {
        Specfield df = new Specfield();
        df.setName(m.group(3));
        df.setLineNumber(lineNo);
```

```
        Matcher mType = DERIVEDFIELD_TYPE.matcher(line);
        if (mType.find()) {
            df.setFieldType(mType.group(7));
        }

        df.setType(Specfield.DERIVEDFIELD);

        if (dt.isDefined(df, symTable)) {
            log(lineNo, "specfield.duplicate", df.getName());
        }

        dt.addSpecfield(df);
    }
}

private void extractSpecfieldRef(DataType dt, int lineNo, String line) {
    Matcher m = SPECFIELD_REF.matcher(line);
    while (m.find()) {
        dt.addSpecfieldRef(lineNo, m.group(2));
    }
}

private void extractClassInfo(DataType dt, List<String> lines) {
    for (String line : lines) {
        processDataType(line, dt);

        if (dt.getName() != null) {
            break;
        }
    }

    for (String line : lines) {
        processParentDataType(line, dt);
    }
}

private void processDataType(String line, DataType dt) {
    Matcher m = DATA_TYPE.matcher(line);
    if (m.find()) {
        dt.setName(m.group(3));
    }
}

private void processParentDataType(String line, DataType dt) {
```

```
    Matcher m = EXTENDS.matcher(line);
    if (m.find()) {
        dt.addParent(m.group(3));
    }
    m = IMPLEMENTS.matcher(line);
    if (m.find()) {
        dt.addParent(m.group(3));
    }
}

private List<DataType> extractInnerClass(DataType dt, List<String> lines) {

    int level = 0;
    int lineNo = 1;
    List<DataType> inners = new LinkedList<>();

    for (String line : lines) {

        Matcher m = DATA_TYPE.matcher(line);
        if (level > 0 && m.find()) {
            InnerDataType innerDt = new InnerDataType(dt.getFileName());
            innerDt.addParent(dt.getName());
            innerDt.setPackageName(dt.getPackageName());
            innerDt.setName(m.group(3));

            dt.addInnerClass(innerDt);

            extractSpecfieldFromInner(innerDt, lines, lineNo);

            int lastLine = findLastLineOfBlock(lineNo, lines);
            innerDt.setEndLine(lastLine);

            extractSpecfieldRefs(innerDt, lines);

            inners.add(innerDt);
        }

        level = (goDown(line)) ? level + 1 : level;
        level = (goUp(line)) ? level - 1 : level;
        lineNo++;
    }

    return inners;
}
```

```
    private void extractSpecfieldFromInner(InnerDataType innerDt, List<String>
        lines, int lineNo) {
        int i = lineNo - 1 - 1;
        while (i > 0 && isComment(lines.get(i))) {
            extractSpecfield(innerDt, i, lines.get(i));

            i--;
        }

        innerDt.setStartLine(i + 2);
    }

    private void extractSpecfieldRefs(InnerDataType dt, List<String> lines) {
        int lineNo = dt.getStartLine();
        boolean comment = false;

        for (int i = lineNo - 1; i < dt.getEndLine(); i++) {

            String l = lines.get(i);

            if (!comment && COMMENT_BEGIN.matcher(l).find()) {
                comment = true;
            }

            if (comment || COMMENT_LINE.matcher(l).find()) {
                extractSpecfieldRef(dt, lineNo, l);
            }

            if (comment && COMMENT_END.matcher(l).find()) {
                comment = false;
            }

            lineNo++;
        }
    }
}
```

Figure A.7 – *SpecfieldCheck* class

A.8 ModifiesCheck.java

```
package be.adrienhoudart.tools.checkspec.checks;

import be.adrienhoudart.tool.domain.Modifier;
import be.adrienhoudart.tool.domain.ModifierType;
import be.adrienhoudart.tools.checkspec.util.ParseUtils;
import static
    be.adrienhoudart.tools.checkspec.util.ParseUtils.cleanMultiLineComment;
import static be.adrienhoudart.tools.checkspec.util.ParseUtils.containsOverride;
import static
    be.adrienhoudart.tools.checkspec.util.ParseUtils.extractPackageName;
import static be.adrienhoudart.tools.checkspec.util.ParseUtils.getBlockContent;
import static
    be.adrienhoudart.tools.checkspec.util.ParseUtils.getPreviousSpecifications;
import static be.adrienhoudart.tools.checkspec.util.ParseUtils.linesToString;
import be.adrienhoudart.tools.checkspec.util.Regexes;
import static be.adrienhoudart.tools.checkspec.util.Regexes.METHOD_SIGN;
import static be.adrienhoudart.tools.checkspec.util.Regexes.MULTI_LINE_EFFECTS;
import static be.adrienhoudart.tools.checkspec.util.Regexes.MULTI_LINE_MODIFIES;
import com.puppycrawl.tools.checkstyle.api.AbstractFileSetCheck;
import java.io.File;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.regex.Matcher;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 *
 * @author aho
 */
public class ModifiesCheck extends AbstractFileSetCheck {

    private static final Log LOG = LogFactory.getLog(
            ModifiesCheck.class);

    private final Map<String, ModifierType> symTable;

    public ModifiesCheck() {
        super();
```

```
    symTable = new LinkedHashMap<>();
    setFileExtensions(new String[]{"java"});
}

@Override
public void beginProcessing(String charset) {
    symTable.clear();
}

@Override
public void finishProcessing() {

    for (ModifierType mt : symTable.values()) {
        getMessageCollector().reset();
        getMessageDispatcher().fireFileStarted(mt.getFileName());

        for (Modifier method : mt.getMethods()) {

            checkModifiesAndEffects(method);

            if (mt.hasImmutableValue() && mt.isImmutable()) {
                checkImmutableClass(method);
            }

            checkModifiersInMethods(mt, method);
        }

        if (mt.hasImmutableValue()) {
            checkSameMutabilityOfItsParents(mt);
        }

        fireErrors(mt.getFileName());
        getMessageDispatcher().fireFileFinished(mt.getFileName());
    }

}

@Override
protected void processFiltered(File file, List<String> lines) {
    ModifierType mt = new ModifierType(file.getAbsolutePath());

    if (ParseUtils.isEnum(lines)) {
        return;
```

```
    }

    String packageName = extractPackageName(lines);
    if (packageName != null) {
        mt.setPackageName(packageName);
    }

    Set<String> parents = ParseUtils.extractParents(lines);
    if (parents != null) {
        mt.addParents(parents);
    }

    String name = ParseUtils.getClassName(lines);
    if (name != null) {
        mt.setName(name);
    } else {
        return;
    }

    mt.setImmutable(ParseUtils.isClassImmutable(lines));
    mt.setDeclarationLineNo(ParseUtils.getClassDeclarationLineNo(lines));

    extractModifyingMethods(mt, lines);

    symTable.put(mt.getName(), mt);
}

////////////////////
// HELPER METHODS
////////////////////

private void extractModifyingMethods(ModifierType mt, List<String> lines) {
    int lineNo = 1;
    boolean comment = false;

    Modifier ref = new Modifier();
    for (String l : lines) {

        if (!comment && ParseUtils.isMethod(mt.getName(), l)) {
            Matcher m = Regexes.METHOD_SIGN.matcher(l);
            m.find();
            String methodName = m.group(3);

            ref.setName(methodName);
```

```
            ref.setLineNo(lineNo);
            ref.setOverride(ParseUtils.containsOverride(lines.get(lineNo -
                2)));

            extractModifiesAndEffects(lines, lineNo, ref);
            extractMethodCalls(lines, lineNo, ref);
            extractInstanceVarAff(lines, lineNo, ref);
        }

        lineNo++;

        if (ref.getName() != null) {
            mt.addModifyingMethods(ref);
            ref = new Modifier();
        }
    }
}

private Modifier extractModifierRef(List<String> lines, int lineNo, String
    l, Modifier ref) {
    Matcher m = MULTI_LINE_MODIFIES.matcher(l);
    if (m.find()) {
        ref.setModifies(cleanMultiLineComment(m.group(3)));
    }

    m = MULTI_LINE_EFFECTS.matcher(l);
    if (m.find()) {
        ref.setEffects(cleanMultiLineComment(m.group(3)));
    }

    String mth = lines.get(lineNo - 1).trim();
    m = METHOD_SIGN.matcher(mth);
    if (m.find()) {
        String visibility = m.group(1);
        ref.setVisibility(visibility);

        String abstr = m.group(2);
        if (abstr != null) {
            ref.setAbstract(true);
        }
    }

    if (containsOverride(lines.get(lineNo - 2))) {
        ref.setOverride(true);
```

```
    }

    return ref;
}

private void extractModifiesAndEffects(List<String> lines, int lineNo,
    Modifier ref) {

    String specs = getPreviousSpecifications(lines, lineNo);

    extractModifierRef(lines, lineNo, specs, ref);
}

private void extractMethodCalls(List<String> lines, int lineNo, Modifier
    ref) {
    List<String> calls = new LinkedList<>();
    List<String> blockContent = getBlockContent(lines, lineNo);
    String content = linesToString(blockContent);

    Matcher m = Regexes.METHOD_CALL.matcher(content);

    while (m.find()) {
        String name = m.group(1);
        calls.add(name);
    }

    for (String call : calls) {
        ref.addModifiers(call);
    }
}

private void extractInstanceVarAff(List<String> lines, int lineNo, Modifier
    ref) {
    List<String> assign = new LinkedList<>();
    List<String> blockContent = getBlockContent(lines, lineNo);
    String content = linesToString(blockContent);

    Matcher m = Regexes.INSTANCE_VAR_AFF.matcher(content);
    while (m.find()) {
        String name = m.group(1) + m.group(2);
        assign.add(name);
    }

    for (String ass : assign) {
```

```
            ref.addModifiers(ass);
        }
    }

    ////////////////////
    // CHECKS METHODS
    ////////////////////

    private void checkModifiesAndEffects(Modifier method) {

        if (!Modifier.PUBLIC.equals(method.getVisibility())) {
            return;
        }

        if ((method.hasEffects() && !method.hasModifies())
                || (method.hasModifies() && !method.hasEffects())) {
            log(method.getLineNo(), "modifies.bothClausesShouldExist",
                method.getName());
        }

    }

    private void checkImmutableClass(Modifier method) {
        if (method.hasModifies()) {
            log(method.getLineNo(), "modifies.mutableClassNoClause",
                method.getName());
        }
    }

    private void checkSameMutabilityOfItsParents(ModifierType mt) {
        for (String parent : mt.getParents()) {
            if (symTable.containsKey(parent)) {
                ModifierType p = symTable.get(parent);

                if (p.hasImmutableValue() &&
                    !p.isImmutable().equals(mt.isImmutable())) {
                    log(mt.getDeclarationLineNo(), "modifies.inheritMutability",
                        mt.getName());
                }
            }
        }
    }

    private void checkModifiersInMethods(ModifierType mt, Modifier method) {
```

```
List<String> calls = method.getModifiers();

boolean isImmutable = mt.hasImmutableValue() && mt.isImmutable();

int nbModifyingCall = 0;

for (String call : calls) {
    if (isImmutable && isModifyingMethod(call)) {
        log(method.getLineNo(), "modifies.callToMutableMethodNotAllowed",
            mt.getName());
    }

    if (isImmutable && isInstanceVariable(call)) {
        log(method.getLineNo(), "modifies.instanceVarAff", mt.getName());
    }

    if (isModifyingMethod(call) || isInstanceVariable(call)) {
        nbModifyingCall++;
    }
}

if (nbModifyingCall > 0 && !method.hasModifies() &&
    Modifier.PUBLIC.equals(method.getVisibility())) {

    if (method.isOverride()) {
        for (String parent : mt.getParents()) {
            if (symTable.containsKey(parent) &&
                symTable.get(parent).hasMethod(method.getName())) {
                Modifier parentMethod =
                    symTable.get(parent).getMethod(method.getName());

                if (!parentMethod.hasModifies()) {
                    log(method.getLineNo(),
                        "modifies.modifiesShouldExistOnParent",
                        mt.getName());
                }
            }
        }
    } else {
        log(method.getLineNo(), "modifies.modifiesShouldExist",
            mt.getName());
    }
}
```

```
    }

    private boolean isModifyingMethod(String call) {
        if (call.startsWith("set") || call.startsWith("add") ||
            call.startsWith("remove")) {
            return true;
        }

        return false;
    }

    private boolean isInstanceVariable(String call) {
        Matcher m = Regexes.SPECFIELD_REF.matcher(call);

        if (m.find()) {
            return true;
        }

        return false;
    }
}
```

Figure A.8 – *ModifiesCheck* class